HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Jarkko Miettinen

SECURITY ASPECTS IN MODERN WEB APPLICATIONS

Thesis submitted for examination for the degree of Master of Science in Technology

Espoo 8.6.2009

Thesis supervisor:

Prof. Eljas Soisalon-Soininen

Thesis instructor:

M.Sc. Kai Virkki

Author: Jarkko Miettinen

Title: Security Aspects in Modern Web Applications

Date: 8.6.2009 Language: English Number of pages: 9+93

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Laboratory of Software Technology Code: T-106

Supervisor: Prof. Eljas Soisalon-Soininen

Instructor: M.Sc. Kai Virkki

Technologies behind the World Wide Web were created initially to ease sharing of static data in form of web pages. Popularity of the Web grew rapidly and led to adoption of web browser as a universal client for application delivery. Though initially inferior to desktop applications, these applications have caught up with their desktop counterparts in features and usability.

These applications, called web applications, use multiple web technologies such as JavaScript and CSS and this multiplicity of web technologies combined with multiplicity of web browsers creates a unique brew of issues not found on the desktop. One of these issues is how data send and used by the applications' users is protected.

In this thesis, security in one mature web application is described and assessed. Such an assessment requires knowledge of information security aspects both in the broader sense concerning all information systems and in the sense of aspects specific to web applications.

Therefore, first introduced are the fundamental concepts of information security, building blocks for all the other sections. The fundamentals are followed by discussion of access control and security aspects in applications. The background part is concluded by discussion of web applications in general and of security questions specific to them.

The latter part explores and applies these theories and methods in a case study of a mature web application. The case study first describes, then evaluates the subject and its security and concludes with discussion of some of the found vulnerabilities and solutions to them.

Although there were some problems in application of security assessment methods, assessment results provided valuable information on the application's weaknesses and improvement proposals. Implementation of the proposals both improved current security and also gave assurance of fewer weaknesses in the future.

Keywords: Web, information security, JavaScript, Ajax

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Jarkko Miettinen

Työn nimi: Tietoturvallisuus web-sovelluksissa

Päivämäärä: 8.6.2009 Kieli: Englanti Sivumäärä: 9+93

Tiedekunta: Elektroniikan, tietoliikenteen ja automaation tiedekunta

Professuuri: Ohjelmistotekniikka Koodi: T-106

Valvoja: Prof. Eljas Soisalon-Soininen

Ohjaaja: FM Kai Virkki

World Wide Webin taustalla olevat tekniikat kehitettiin alun perin helpottamaan tiedon jakamista. Tämä jaettu tieto oli aluksi muuttumatonta tai harvoin muuttuvaa, mutta webin yleistyminen muutti tilanteen. Yleistyminen teki web-selaimesta nopeasti yleismaailmallisen ohjelmiston sovellusten tuottamiselle ja käyttäjälle välittämiselle. Vaikka nämä web-sovelluksiksi kutsuttavat ohjelmistot olivat alkujaan työpöytäsovelluksia monin tavoin huonompia, muuttui tilanne nopeasti.

Web-sovelluksissa käytettävät tekniikat, kuten JavaScript ja CSS, sekä web-selainten moninaisuus muodostavat yhdessä erinäisten kysymysten sekamelskan, jota vastaavaa ei työpöytäsovelluksissa ole. Eräs tärkeä kysymys on, miten sovellusten käyttäjien lähettämä ja käyttämä tieto turvataan.

Tässä diplomityössä tutkitaan ja kuvataan erään web-sovelluksen tietoturvallisuutta ja tietoturvaratkaisuja. Arvioiminen vaatii tietämystä sekä yleisistä tietoturvallisuuskysymyksistä että erityisesti web-sovelluksiin liittyvistä kysymyksistä.

Ensimmäisenä tutustutaan tietoturvallisuuden peruskysymyksiin ja käsitteisiin, joiden ymmärtäminen on välttämätöntä. Perusteiden jälkeen käsitellään pääsynhallintaa ja sovellusohjelmistojen tietoturvallisuutta. Ensimmäinen osa päättyy web-sovellusten ja niihin liittyvien tietoturvallisuuskysymysten esittelyyn. Jälkimmäinen osa diplomityöstä soveltaa käsiteltyjä teorioita ja menetelmiä erään web-sovelluksen tapaustutkimuksessa. Tapaustutkimuksessa kuvataan ja arvioidaan sovelluksen tietoturvallisuutta sekä lopuksi esitellään löydettyjä haavoittuvuuksia ja ratkaisuja näihin haavoittuvuuksiin.

Vaikka joidenkin ohjelmistojen tietoturvallisuuden arviointimenetelmien soveltamisessa olikin ongelmia, saatiin tapaustutkimuksen tuloksena tärkeää tietoa heikkouksista ohjelmiston tietoturvallisuudessa ja hyviä esityksiä näiden heikkouksen poistamiseksi. Esitykset toteuttamalla parannettiin sekä nykyistä tietoturvallisuutta että vakuututtiin siitä, että heikkouksia esintyy jatkossa vähemmän.

Avainsanat: Web, information security, JavaScript, Ajax

Preface

This work, done at Efecte Corporation, describes my journey from a neophyte to a beginner in web security.

I would like to thank professor Eljas Soisalon-Soininen for the good supervision he provided. Without the help of my instructor, Kai Virkki, this work would not have been completed. A big thank-you to Kai for the countless hours he has spent reading this thesis and for the countless suggestions he provided.

I also wish to thank Lauri Lehmijoki and Minna Torniainen for the proofreading parts of this thesis and the suggestions they have given me.

A special thanks go to my friends, siblings and co-workers for the fun and help they have provided me.

Lastly, I would like to thank my parents for being always so supportive.

Otaniemi, 8.6.2009

Jarkko Miettinen

Contents

Glossary of terms and abbreviations

User Agent	A client application for some network protocol, mainly used as a technical term for web browser.
WWW	World wide web, a set of interlinked documents accessed through Internet.
URI	Uniform resource identifier is a character string used to identify resources in the Internet.
Resource	Something identified by an URI in the Internet, such as a document or a person.
Site	Short form of website, collection of related documents hosted on a web server.
HTTP	HyperText Transfer Protocol is a protocol used to transmit documents in WWW.
PRNG	Pseudorandom number generator is an algorithm for generating numbers that approximate random numbers' properties.
Hostname	A unique name for a device connected to a network, usually to Internet.
NTLM	NT LAN Manager is a authentication protocol by Microsoft commonly used in Windows networks.
XML	Extensible Markup Language is a specification for creating markup languages. Used by different applications as a basis for their data format.

Part I
Background

1 Introduction

World-wide web, familiarly known as WWW, is an inter-linked system of hypertext documents that is accessed through Internet. WWW was conceived 1989-1990 by an Englishman Tim Berners-Lee working at CERN for physicists to share data and first versions of needed tools were released by Christmas 1990. These tools included HyperText Markup Language to create the web pages, HyperText Transfer Protocol to transfer those pages, and web server to serve those pages to web browsers. What followed was enormous growth of both information available and users in the Internet.

Although web pages were originally just static pieces of data served by web servers, it did not take long until web browser was discovered as the universal thin-client. This led to creation of *web applications*, distinguished from normal desktop applications by two facts: they required no installation as they were delivered over the Internet on-demand and work on many platforms thanks to requiring only a standards-compliant web browser. At first, web applications did all the work on the web server and thus had lower responsiveness and usability than their desktop counterparts. Such downsides were, in any event, temporary as faster computers and new web technologies brought web applications' usability and responsiveness to the level of desktop applications.

Moving applications from desktop to web created new problems, especially on security: generally everyone can access websites and web applications, whereas desktop applications are accessible only by users that otherwise have access to the computer running the application. Main difference is that while users of desktop applications can only harm themselves by breaking a desktop application, malignant web application users can potentially cause harm to web application provider and to other users. Thus web application developers need to consider security aspects that are often unnecessary on desktop applications. Luckily, these security aspects are nothing new, information security researchers have studied the same questions for decades, only their manifestations.

1.1 Goal and scope of thesis

The primary goals of this thesis are evaluation and improvement of security of one specific web application named Efecte. A secondary goal is to create a document containing introduction to security concepts needed in web application development. This secondary goal describes the scope of this thesis: we will not review the whole vast information security research but only the research that is directly useful in

secure web application development, mainly from computer security.

1.2 Structure of the thesis

The structure of the thesis is following:

In Section 2, basic concepts and problems of information security are introduced. This includes defining what is a "secure" system and how we can be assured of its security.

In Section 3, main theme is building secure software. First introduced and reviewed are different models of access control. Discussed are both the general methods of access control and different models for it. Next discussed are both problems and solutions for building secure software followed by methods to find weaknesses and vulnerabilities in software and its design

In Section 4, core technologies and concepts behind the Web are introduced. Additionally, architecture and vulnerabilities common in web applications are described and discussed.

In Section 5, gives a brief introduction to Efecte corporation and to the case-study subject, a web application named Efecte. Architecture and concepts of Efecte are described first, followed by description of Efecte's security. After this groundwork has been laid down, Efecte's current security is analyzed. The analysis starts from high-level view proceeding to review of select parts of Efecte's source code.

In Section 6 focuses on selected problems found in the security analysis. The problems are described and evaluated and mitigation efforts taken are discussed.

Finally, in Section 7 conclusions are drawn and some future improvements not mentioned earlier are discussed.

2 Information security

Information, meaning messages and saved records, might be the singular reason why civilization exists: each generation does not need to do the same mistakes as their elders did because they can learn from them. Yet most information stored nowadays is something completely different from instructions of how to make a fire: census data, subscriber records of a magazine, phone calls, photographs and all the other things dispensable for survival of the civilization. These dispensable pieces of information are often such that we would not want to share them with everyone nor would we want that everyone could change and edit them: one would not want that after one has paid for a magazine subscription, someone changes one's name and address to her own and starts to receive magazines one has paid for.

Information security provides means to protect your magazine subscription data and other important data. Paraphrasing US code, information security means protecting information and information system from unauthorized access, disclosure, modification, destruction or disruption in order to provide *confidentiality, integrity and availability*. To be able to discuss information security, the basic concepts and vocabulary must be first clearly defined. This chapter starts with an introduction to confidentiality, integrity and availability, followed by the rest of the information security core concepts.

2.1 CIA triad

2.1.1 Confidentiality

Confidentiality means concealment of data, concealment of resources and concealment of existence of such data and resources. Reasons for keeping information secret are manifold: laws and regulations may force secrecy, information may have commercial value or a private citizen may just want to keep her data private. Rise of computers in such sensitive fields as military, government, financial-, and health-sector gave incentive to develop security models that provide confidentiality. Formal work on information security models (see 3.2) was first motivated by and modeled after *"need to know"*-principle common in military and government. "Need to know"-principle states that having high enough security clearance is not enough to allow a person to see some secret information — the person has to demonstrate a need for such information.

Confidentiality requirements also often apply to existence of the data as a statute ordering patient records to be confidential but leaving the existence of such record public would very much undermine the statutes spirit. What disease you actually had when you visited venereal diseases center may not matter that much if your whole neighborhood knows about your visit as you have some record there. The information protected need not reside in databases or files: knowledge resources an organization uses is information that the organization often would like to keep secret

because it could help other attacks, as is the case with knowing the IT-infrastructure of an organization, or give competitors clues on what sort of services could be behind its success.

Confidentiality is provided by a combination of access control mechanisms and trust. Common access control mechanisms include cryptography (see HTTPS, 4.1.2) and access controls provided by computer systems, discussed in Section 3.1. All choices to provide confidentiality, as with other parts of security, make some assumptions about possible attackers to the system and resources they have on their disposal. These assumptions and trust on implemented access controls thus underlie the confidentiality of a system.

2.1.2 Integrity

Improper tampering or unauthorized changes reduce the trustworthiness and therefore usefulness of information. Integrity refers to this trustworthiness and includes both integrity of the data and integrity of the origin of the data. Data integrity is the internal integrity of this data: has the data been modified by unauthorized entities or has some of the data been lost. Origin integrity refers to trustworthiness of the data source: how much trust can be placed on data from this source; how credible and accurate information the source gives. Integrity is quite different from confidentiality; confidentiality of the data is a Boolean proposition whereas integrity concerns both data protection in current storage and the origin of the data.

There are two classes of integrity mechanisms: prevention mechanisms that block unauthorized changes to the data and detection mechanisms that detect untrustworthy changes in the data. Prevention mechanisms can either prevent only changing of data by unauthorized parties or prevent changing of data in unauthorized ways. Examples of such are database integrity mechanisms that require that a specific value cannot be `NULL` or needs to be unique. Detection mechanisms do not try to prevent improper and unauthorized changes in the data but instead report that data integrity has been lost. These mechanisms may analyze the data to see if the required constraints still hold (a bank account must have a non-negative balance), to calculate of checksum on the data (such as cryptographic hash values of a signed executable file) or analyze system logs to see if unauthorized actions were taken.

Proper evaluation of information integrity is difficult as the origin of the data needs to be included in the considerations.

2.1.3 Availability

Both integrity and confidentiality usually have little of use if the data is not available; a way to ensure confidentiality and integrity of records would be to drop them to the bottom of the ocean floor but this would make the data hard to use, that is, reduce its availability.

Accidents and deliberate attacks both affect availability: an important computer

serving data can accidentally be shut down or attackers may launch an attack to deny users access to a specific computer or service. Protecting availability begins with a model of the expected use that is then used to decide what mechanisms are put in place to ensure the availability. For a web server this model could include that maximum amount of normal users and in a case of an attack, the bandwidth available to attackers. Should the situation change in a way that model is no longer correct, such as the attacker being able to use company's internal network, the mechanisms based on the model can fail and result a loss of availability.

Attacks on availability can be hard to distinguish from innocent but atypical usage. For example, a public web server may get attention from more popular sites and as a consequence get much more traffic than normally, having the same effect on the server as a contrived *denial-of-service* attack. This can lead to false alarms with server administrator hastily thinking that the server is under attack even though there has just been a link to a page on the server on a popular link aggregation site. Additionally in the public web server case, availability can be the most important factor if the revenue of a company is based on advertisements served on each web page. Server may not even contain any confidential information and integrity problems do not cut the revenue as fast as availability problems.

2.1.4 Conclusion

Together confidentiality, integrity and availability form the so-called CIA triad and they have long been held as the core concepts and goals of information security. Depending on the system, acts that augment one of the goals can affect others. For example, Wikipedia-styled system where everyone can edit any data in the system can arguably boost the integrity of this data but makes confidentiality all but impossible.

CIA triad gives us basic concepts for information security but to be able to build secure systems and assess their security, more concepts are needed — starting from definition of what is "secure".

2.2 Rest of core concepts

2.2.1 Security policies

Security policy defines what is *secure* for a system. Security policy states what is and what is not allowed in a system, be it an organization or a computer. To implement the policy, other means are usually required. These means are security model and security mechanisms. Security model formalizes and specifies the policy and security mechanisms enforce the model and policy. Security mechanism can be anything ranging from a process to an automatic tool that enforces at least part of the security policy. A security mechanism is said to be either *secure* if all states the mechanism allows are secure, *precise* if the mechanism only allows all secure states

or *broad* if the mechanism allows also insecure states.

To illustrate the difference between security policy and mechanism, consider security policy in a company that forbids reading of other employees emails except by the manager of the employees in case the employee is on a holiday and an important email is expected. If Mike, a system administrator in such a company, having unlimited access to email servers, read Peter's, who is another employee in the company, email he would violate the security policy even though there would be no security mechanism in place to enforce this policy. Then again, if Annette, Peter's manager, knew that Peter will receive an important mail about call for bids within next few weeks and Peter is at the same time is in Tahiti on honeymoon, she could read Peter's email for the next few weeks. This would probably be done with the aforementioned Mike's help and neither of them would breach the security policy provided that Mike would not actually read the mail but just help Annette to read it. But what if Annette knew, or thought she knew that Peter would receive an offer of employment from a competitor within the next few weeks?

In the example the policy was simple but still had some uncertainty about in what cases Annette was allowed to read the mail and in which cases she was not. Security policies are often formulated in a natural language known for its ambiguousness. Furthermore they are not formulated precisely and exist in a world with other policies, laws and regulations. It is then no wonder that policies often are not clear cut. This manifests in cases which are not clearly allowed or forbidden and require interpretation of the security policy instead of straightforward inference. The problem is even more clear when two or more entities with security policies communicate: the policies are not usually exactly the same, leading to either breaches of policy or more difficult communications when party A cannot send data to party C because C's security policy is different and does not protect some part of the data.

2.2.2 Assumptions, trust and assurance

Security policy determines what a system should do and how to do it. Security of the system rests on *assumptions* that are specific to what kind of security is required and to the environment the system is used in. Paraphrasing [Bis02], assume that opening a door lock requires a key and that the lock is secure against lock picking. This assumption is based on the fact that most people cannot pick locks. However, a good lock picker can open the lock without the key. Thus, in a environment with a skilled but *untrustworthy* lock picker, security of the system lays on invalid assumption and cannot be considered secure.

Were the lock picker *trustworthy*, the assumption would be valid and system could be secure if also other assumptions were valid. The term trustworthy means that the lock picker will not pick the lock unless authorized by the owner of the lock. Thus the lock picker can be *trusted*. In essence, trusted means the same as something that can hurt you if untrustworthy.

Trusting the system to be secure requires trusting that security policy correctly

and clearly partitions system's states to "secure" and "insecure" and that security mechanisms prevent the system from entering a "insecure" state, meaning that union of security mechanisms is a security mechanism that is not broad.

This trust cannot be quantified precisely and instead we have to estimate how much we trust the system. We can implement specific actions that *assure* us of the system's trustworthiness, such as arguments or proofs that the system's design and implementation satisfy system's specification or tests and evaluations of the system that did not find any undesired behavior (See section 3.3). Assurance is then our estimate, that is, subjective probability that the system in question will not fail in some specific way.

Quote from Lampson et al. [LABW92] summarizes assumptions and trust:

> In any security system there are assumptions about authority and trust. The theory tells you how to state them precisely and what the rules are for working out their consequences. Once YOU have done this, You can look at the assumptions, rules, and consequences and decide whether you like them. If so, you have a clear record of how you got to where you are. If not, you can figure out what went wrong and change it.

2.2.3 Threats and attacks

A potential violation of security is called a *threat*; security violations do not need a realization to be called as threats. Actions that can cause realization of threats through *vulnerabilities* are called *attacks* and executers of such actions called *attackers*.

Vulnerability is a weakness in the security system that can be exploited to cause harm.

Threat is a potential violation of security policy, usually to steal or damage assets.

Attack is an action that can realize a threat.

Control mechanism is a protective measure such as action, device, procedure or technique that reduces or completely removes a vulnerability. Often called just "controls".

Pfleeger and Pfleeger summarize this neatly: "A threat is blocked by control of a vulnerability" [PP06].

As an important part of information security is creating and using different controls, types of threats that they control need to be well understood. Combining categorizations by Gollman [Gol05] and Bishop [Bis02] we get five different classes of threats.

- *Snooping* or *wiretapping* causes information disclosure. Snooping is a passive attack where attacker listens to communications that are not intended for her or browses through files or other information that is available but are not meant for the attacker. Confidentiality services, such as encryption or more broadly, access control, is used to counter this attack.

- In identity *spoofing* the attacker pretends to be another person or computer. Spoofing is also known as *masquerading*. Such an attack can happen, for example, when a user tries to access a computer across the Internet but another computer masquerades as the target computer. The user using this computer can then receive wrong and deceptive information and if the user authenticates, attacker may get the credentials of the user. This threat is countered by integrity mechanisms, namely by authentication of both the user and the system.

- *Tampering* with data, that is, unauthorized *alteration* of the data, is an active attack. Tampering attacks can lead to various results such as denial of service or information disclosure. Denial of service can happen when a system uses tampered data to determine what action to take, for example a web service could be shut down by giving it specific parameter that is not available through its web page based interface. *Man-in-the-middle* attack in an example of information disclosure: when two entities communicate with each other, a third party spoofs to both parties as the other party and can read and tamper their messages as long as they have not agreed on encryption key beforehand or do not use a trusted third party for identification. Integrity services can be used to counter these attacks; Man-in-the-middle attack, for example, is thwarted by public key infrastructure.

- *Repudiation* attack is a false denial of having done something such as having sent or received a message. Repudiation is an important concern in web commerce where a customer may deny that she has ordered anything and therefore stop her credit card company from charging her for her purchase. Integrity mechanisms are used to combat repudiation: having the buyer enter information that only she should know such as her credit card number and precise address gives the seller and credit card company assurance that card owner is the actual orderer.

- *Delay* is an attack that makes a service, such as a website, partially or completely unavailable (*Denial of service*). Delay attacks require some way to control either the service structure or the medium used to contact this service (e.g. a network). Delay attacks can enable other attacks such as spoofing: if the attacker cannot spoof as the main server of some service but can spoof as some backup-server, the attacker can drive users to the spoofed server by inducing delays to the main server. Availability mechanisms, such as blacklisting offensive users, counter this threat.

2.2.4 Risks and risk analysis

A moment of contemplation on the threat classes described in Section 2.2.3 allows anyone to come up with multiple threats against any system they know if they just let their imagination run free to come up various kinds of ideas. The question is: how many of these ideas then are "crazy"?

It is usually easy to come up with so many threats against the system that they all cannot be protected against. Realization of some threats is more serious than of other threats (cause more damage measured in some suitable unit such as money) and some threats are more probable or easier to realize. When choosing what control mechanisms to implement, a rational decision-maker should take these both facts into account and then choose the action that *mitigates* the risk the best according to a cost-benefit analysis.

Different sources have a bit different uses for risk analysis vocabulary. Following usage of Richard E. Fairley [Fai05], we define terms for risk analysis as follows:

Risk is the probability of realization of a threat.

Risk impact is the loss caused by a realized threat.

Risk exposure is the expected value of a threat: risk of the threat multiplied by risk impact of the same threat.

Risk mitigation is a course of action that can reduce the probability, loss or both associated with a certain threat.

Risk mitigating actions are divided into four main categories: actions that *avoid* the risk, actions that *reduce* the risk, actions that *transfer* the risk and finally, actions that *assume* the risk. Cost-benefit analysis on choosing what risks to mitigate and how is based on comparing *risk leverages*. Risk leverage is

$$\frac{(\text{risk exposure before mitigation}) - (\text{risk exposure after mitigation})}{(\text{cost of risk mitigation})}$$

Now, we have all the concepts needed for a basic risk analysis. The outline of a risk analysis is listed below.

1. Identify the assets. Before we can do anything at all, we first need to decide what to protect. This is usually done when security policy is formulated.

2. Determine vulnerabilities. Here we start with the CIA triad and want to consider situations that could cause loss of confidentiality, integrity or availability. There is not simple process or checklist to find all vulnerabilities and many organizations have come up with their own processes such as VAM methodology by RAND Corporation [AAMS03] and RMF by Cigital [McG06]. We will return on this question in Section 3.4.

3. Estimate the likelihood of exploitation. This requires considering stringency of current controls and estimating likelihood for evading the controls. Estimating likelihood can be the hardest step in the analysis.

4. Compute expected loss. Determining value of physical object can seem quite straightforward as we only have to compute replacement costs of the object: cost of a new object and cost of work to replace it.

5. Survey the control mechanisms and their costs. Controls have both positive and negative effects: saving document after each revision improves integrity of the system but consumes more resources and may make the system harder to use. Controls do not need to be chosen through a rigorous process when there are only a few available. Otherwise it is wiser to use a more systematic approach such as the aforementioned VAM.

6. Compute risk leverages and decide the best course of action. After calculating risk leverages for each control and for doing nothing, we can do a cost-benefit analysis and choose the best course of action to mitigate risk.

Although the risk analysis method described here is intended for information security, it is generally not restricted to information security only.

2.2.5 Conclusion

Security policy defines when a system is considered secure. Security of every system is based on some set of assumptions that, if unfounded, can make the system insecure. To be able to trust a system to be secure, we need to have some level of assurance for its security. This assurance comes in form of proofs and arguments that support the fact that the system is working as intended by security policy.

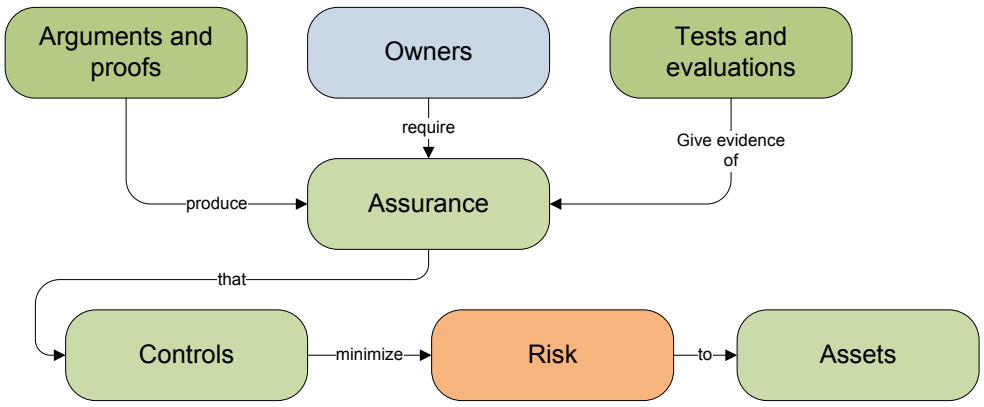


Figure 1: Concepts affected and affecting Assurance. Adapted from [ISO]

.

Uncontrolled vulnerabilities give rise to threats and attacks. Risk is a measure of how likely a threat is realized. These relations are described in Figure 2.

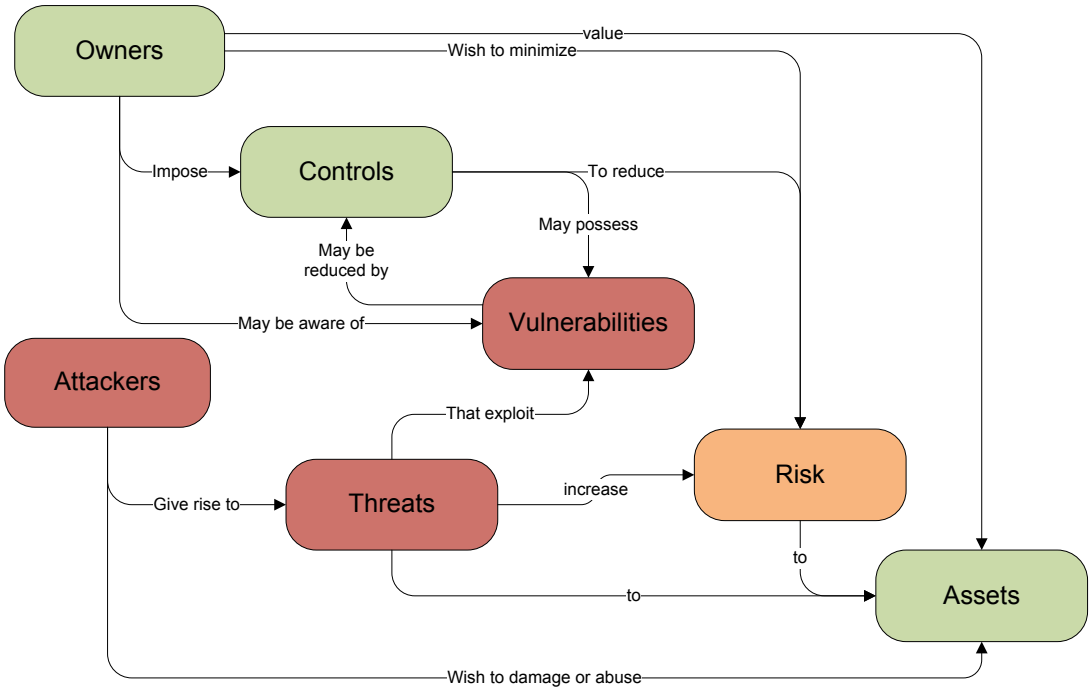


Figure 2: Relations of some of the concepts introduced in this section. Adapted from [ISO]

.

2.3 Human factor

Implementing a policy becomes the harder the larger the organization the policy is imposed on. Incorrectly implementing a security policy may render it useless or even harmful to the organization. Most important problem in security policies is that they cannot create anything that boosts a company's bottom line; the policies can only prevent losses. Therefore policies will not gather respect unless they can be shown to have prevented some specific loss or there have been losses before because of the bad security practices. The immediate effect may even be negative as the policies often hinder user productivity.

An additional problem may be that those who are responsible for security policy implementation may not have enough power or resources to actually implement them. As always, responsibility without power and vice versa leads to problems, as those who are responsible and see the security problems need to petition managers and other people to have the policies actually implemented.

In the case of sufficient power, resources may still be lacking. For example, consider a case where the policies are implemented using computers and the person responsible for this is the system administrator. Although she already has access to company's computers and therefore power to establish the policy, she has dozens of other responsibilities, many of which are more visible such as helping other employees with their computer problems and updating company website. This can

lead to sloppy implementation as the security responsibility is seen as secondary to the other responsibilities. All above have assumed that there is already a sane security policy that only needs implementation but if the security policy needs to be designed based on vague requirements, designing and implementing security policy needs money and time, both unfortunately scarce resources.

The weakest link in many security systems are the people using it. This is especially true in computer security systems as many controls can be surpassed by human intervention. Lack of training and honest mistakes can both significantly weaken security systems. Lack of training can cause security policies, such as verification of backups, not to be followed. Honest mistakes are the worse the more power over security policies is wielded by the mistaken person: incorrectly implemented password policy may cause the entire organization to have weak passwords and therefore to be an easier target for attacks.

Attacks can come both from in- and outside the organization. Earlier, insiders were responsible of 73 to 90 percent of computer security crimes [Uni99], but this has changed. With the rise of the Internet and network services, insider's share of attacks has decreased significantly and now most of the attacks come from outside, while 44% of attacks are performed by insiders [Ric].

Insiders have the advantage of knowing the access controls and people better so circumventing the first and persuading the latter is easier. Knowing both official organization and unofficial organization, insiders have a better chance of avoiding detection and launching a successful *social engineering* attack. Social engineering is a technique where attacker induces someone to give up secrets, such as their password by bribing, lying or purporting to be someone else such as a senior officer of a company. One last thing making insider attacks more dangerous is that many organizations have hard walls of security against the outside but inside these walls there is little security.

Outsiders do not have the informational advantage the insiders have but they can partly make this up with their other advantages. Determined attackers from the outside are usually much more skillful technically for attacking is their area of expertise and attacking through the Internet, they can more easily cover their tracks in case the attack is detected.

2.4 Conclusion

Contrary to what recent scares of terrorism, computer viruses and killer flus would make us believe, security is always a trade-off. All security is based on some assumptions that may prove to be wrong and these assumptions hold the key to understanding a system's security. Only if assumptions are correct and trust is well placed, can a system be secure. The concepts introduced here make up the vocabulary needed to discuss and determine these assumptions and control mechanisms that enable creation of trusted systems.

3 Computer security

Whereas Section 2 was an introduction to general concepts of information security, this section will focus more on concepts and methods needed for building secure computer systems such as web applications. Computer security being a large field, only subjects connected to secure applications, especially web applications, are discussed. This section first introduces different access control models, models to control who is allowed to do what with data. This is followed by a quick review on fundamentals of building secure software. Lastly, we discuss methods to assess vulnerability of software to different attacks.

3.1 Access control

Ability to allow or deny access to different resources based on the accessing entity is called access control. Access control consists of *authentication* ("Who is accessing the system?"), *authorization* ("Can she do this?") and possibly of *auditing* ("Check what she has done"). Security policies and therefore security models that model those policies can use two kinds of access control, either alone or in combination. First one is discretionary access control (DAC) where the owner of the data decides who can access the data. In the other one, mandatory access control (MAC), some higher authority controls the access and owner of the data cannot override it.

We will first define access control in the general sense and describe a general model that forms the basis for other models. Later, different mandatory access control models are introduced, starting from Ball-La Padula confidentiality model, ending to role-based access control and discussion of special access controls.

3.1.1 Basic access control model

Most computer systems that implement access control, use variant of *access control matrix model* first proposed by Butler Lampson [Lam71]. Elements of this model are shown in Figure3.

In the model, authentication deals with obtaining the source of request which maps to a *principal*. Noting ambiguousness of term principal [Gol01], we define principal here as "an entity that can be granted access to objects or can make *statements* affecting access control decisions" ([GGKL89]). These statements a principal makes are primitive *access operations* such as "open file foo" or "execute program bar" and it is these primitive operations that *reference monitor* then either allows or denies.

In addition to this authorization, reference monitor may also create an audit trail by logging all or some access operations made (not included in Lampson's model but in modern operating systems based on it). This trail can later be audited to see that no security violations have taken place or to find culprits. Finally after these checks, reference monitor allows access to resources principal sought. These resources are called *objects* and usually include files, different users, memory addresses and such.

There is also a specific sub-class of objects called *subjects* that are active entities within the system (e.g. a running process). For purposes of reference monitor, subjects are bound to principals and monitor makes decisions based on access rights of the principal.

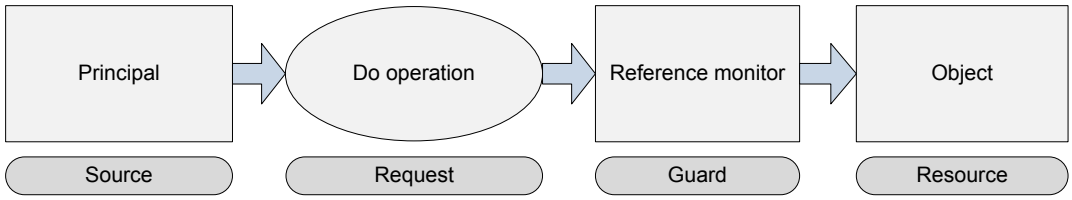


Figure 3: The fundamental access control model

3.1.2 Authentication

The first problem faced in access control model is authentication. Authentication is the process of binding an external entity to a principal. In order to authenticate, this entity must provide her alleged identity (commonly user name) and then some additional information that can prove the identity. This additional data comes from one or more of the sources listed in Table 1, most commonly from a password (knowledge).

Table 1: Four different ways to authenticate

Knowledge	Something the user knows, such as a password
Biometric	Something the user is or inherently has, such as fingerprints or an iris image
Item	Something the user has, such as a key
Location	Where the user is, such as in the front of the terminal

Each of these authentication methods has its own strengths and weaknesses:

Knowledge such as a password is theoretically the most secure as it only exists within the head of a principal, but the principal can choose weak passwords [Kle90, YBAG04] and can be subjected to physical and psychological coercion.

Biometric identifiers such as iris images or fingerprints are always available and can be hard to fake or steal. As such biometric identifiers rely on statistical methods, they are subject to cases where legitimate users are not recognized (false negative) and to cases where illegitimate users are identified as a legitimate user (false positive). For a more throughout discussion, see Anderson [And01].

Items such as tamper resistant security tokens are physical keys and share some problems of other physical items as they too can be lost and stolen.

Location trades authentication problem for physical security problem: how to keep people out of a specific location. This trade can be quite beneficial as physical security is better understood but has the downside that using computer system only from a specific location can be very cumbersome.

Password-based authentication is the most common authentication method probably because it is the easiest to implement and it can be quite strong authentication method given some assumptions, such as users do not write passwords down and that they use strong password, hold. Most text books on computer security contain good advice on password systems, see, for example, [PP06, Gol05].

In order for authentication to be considered "strong", at least two of the factors listed in Table 1 need to be used. The combination most often used is knowledge, namely a password, combined with some other source of information such as a security token.

3.1.3 Authorization

Authorization is, as stated earlier, answering the question "is subject allowed to do this?". As subjects are mapped to principals, what needs to be considered in authorization is only tuple (s, o, a), with

$s \in S$, where s is a specific subject and S is set of all subjects in a system,

$o \in O$, where o is a specific object and O is set of all objects in a system, and

$a \in A$, where a is a specific access operation and A is set of all access operations in a system.

Here the access operations vary from system to system but commonly include at least reading the object (commonly abbreviated as "r"), writing to it ("w"), executing the object as a program ("x") and changing access rights of the object ("c").

For each pair $\{s, o\}$, entry of an access control matrix $M_{s,o}$ (see Table 2(a)) contains set $\tilde{A}$ denoting the subject's allowed access operations to the object, called *access rights*. If $a \in \tilde{A}$, then a is allowed, otherwise it is denied. If owners of different objects are allowed to change access rights of different subjects, we get discretionary access control, otherwise mandatory access control. The policy behind authorization is the key difference between different access control models and will be discussed further in Section 3.2.

As present computer systems contain millions of files and multi-user system can have thousand of different users, storing and accessing all these rights in a access control matrix (as per Table 2(a)) would be wasteful and in large systems, perhaps impossible. Instead, access rights can be stored per subject in capabilities (Table 2(c)) or per object in access control lists (ACLs, Table 2(b)) so empty access rights need not to be stored at all.

Table 2: Different modes of access control. r = read, w = write, x = execute

(a) Access control matrix

Subject	shared.txt	program.exe	diary.txt
Alice	{r, w}	{r, x}	∅
Bob	{r}	∅	{w}

(b) Access Control Lists

Object	Subjects and access rights
shared.txt	Alice: {r,w}; Bob: {r}
program.exe	Alice: {r,x}
diary.txt:	Bob: {w}

(c) Capabilities

Alice	shared.txt: {r, w}; program.exe: {r, x}
Bob	shared.txt: {r}; diary.txt: {w}

Difference between capability- and ACL-based systems is not limited to storage mode. In addition, capabilities and ACLs differ in how access rights are granted and propagated. As seen in Table 3, ACL-based system are object-centric whereas capability-based systems are subject-centric.

One of the key features in capability-based system is ability to give other subjects just the access rights that are needed and nothing more. This is known as the *principle of least privilege* (see Section 3.3.3). Using capabilities could prevent a common source of security problems called *confused deputy* [Har88] where a subject, usually a computer program, is fooled by some other party to use its own access rights instead of the other party's rights. For example, a web service is supposed to display only files readable by its user but instead uses its own access rights to determine what is readable, allowing users to read files they are not supposed to access.

Despite its benefits, systems are not often based on capabilities but on ACLs: all contemporary mainstream operating systems use ACLs whereas capabilities are limited to research operating systems such as KeyKOS and to implementation at application level.

3.1.4 Accountability and Audit

Logging every access, successful or not, to *audit log* seems appealing as it allows evaluation of all actions that could affect system's security. As good as it sounds, total logging has two big problems: such a log would usually be enormous and reduce system's performance and finding useful information in such data would be like searching a needle in a haystack.

On author's single-user desktop system, opening Microsoft Word caused over 1100

Table 3: Similarities and differences in ACL- and capability-based systems.

ACL	Capabilities
All subjects accessing an object need to authenticated.	System only checks for valid capability, it does not need to authenticate subjects.
System maintains ACL for each object.	Forging capabilities must be hard or impossible.
System mediates all changes to ACLs to protected their integrity.	Owner of an object can give access rights to other subject without system's help.
Only subjects allowed to change object's ACL can grant access rights.	A capability can be passed to others in part or in full (delegation).
Access rights to a certain object can be revoked centrally.	Each subject can only revoke its own capabilities - access to a specific object may not be centrally revocable.
It is easy to see who have access to a specific object but arduous to see what objects a specific subject can access as it requires enumeration of all objects in the system.	It is easy to see what a specific subject can access and instead it is laborious to see access rights to a specific object.

accesses to file system and registry and during one minute of just writing this document and listening to music, over 40,000 such accesses were recorded. Logging such data for 24 hours would create a log with over 55 million entries so it is not hard to imagine how loaded larger systems with more users and activity would be because of such logging. Usually the amount legitimate of access overshadows that of attacks making detection of such attacks hard - especially as system administrators cannot go through the logs manually but have to create some programs to analyze the logs and detect intrusions and intrusion attempts.

It is clear that less amount of logging is needed but how much? If two employees write to the same order file and order is later discovered illegitimate, holding both *accountable* for the order may be wrong but if logging is only done on level of writing and closing files, nothing more can be done. Still, in many other cases, logging every character written to a file is too detailed and leads to enormous amount of logging.

3.2 Security models

3.2.1 Bell-La Padula model

Bell-La Padula[1] model [BL73], BLP for short, is a seminal model of confidentiality and has influenced many other security models. BLP takes mandatory access control of security levels and categories, combining it with discretionary access control of "need to know"-principle.

In BLP, each subject has security clearance $C(s) = l_s$ and each object a security classification $C(o) = l_o$. Additionally, each object has a set of *categories* it belongs to and each subject has a set of categories it has access to. These categories can, for example, denote in what projects the subject works. In such scenario, object with multiple categories would mean that the object is used in multiple projects.

These two different mandatory access controls, set of classifications C and set of categories K form a set of *compartments* or *security levels* L, $L = C \times K$. Define the relation *dom* (dominates) as follows.

Domination: Security Level (C_1, K_1) dominates (*dom*) security level (C_2, K_2) if and only if $C_2 \leq C_1$ and $K_2 \subseteq K_1$.

Now set of security levels L together with relation *dom* form a lattice.

Last thing needed to state BLP model is access matrix M. $M = (M_{so})_{s \in S, o \in O}$ gives access permission that given subject s has on the given object o, based on subject current "need to know". Security Levels together with access matrix allow the definition of BLP security policies.

Simple Security Condition: Subject s can read object o if and only if s dom o and M_{so} contains read access.

Simple security condition simply states that subject needs to have both access through mandatory access control and through discretionary access control. It is usually stated as "no reads up".

$*$-Property: Subject s can write to object o if and only if o *dom* s and M_{so} contains write access.

$*$-property prevents information leakage to lower security levels as information can only be written to current security level or to levels that dominate the current level. This property is informally stated as "no writes down." Without additional mechanisms, $*$-property would prevent every subject s from communicating with subjects $\widetilde{S}$ for which $L(s)$ dom $L(\widetilde{S})$.

To allow this sort of communication from s to $\tilde{s}$ s can decrease its *effective security level* from (C_s, K_s) to $(C_{\hat{s}}, K_{\hat{s}})$ such that (C_s, K_s) dom $(C_{\hat{s}}, K_{\hat{s}}) \wedge (C_{\tilde{s}}, K_{\tilde{s}})$ dom $(C_{\hat{s}}, K_{\hat{s}})$.

[1] Leonard La Padula is often misspelled as LaPadula, for example in the cover of [BL73].

As s temporarily loses some of its access rights, it cannot pass information to subjects with lower clearance. While it is not realistic to expect a human to forget all information seen from higher levels, the model is intended for computer systems: a process may lower its security clearance and lose access to some objects, such as files, in order to edit objects with lower security clearance.

Bell-La Padula model is a simple but effective model, roughly abbreviated as "no write down, no read up". It provides confidentiality in environments where confidentiality is most highly valued of the CIA triad, not touching questions of integrity and availability in any way. Strong confidentiality makes it suited for governments and military but in many businesses data integrity is more important.

3.2.2 Integrity models

Whereas in military environments, access to information is controlled by both security clearance and "need to know", in commercial environments an individual is usually given the data if she just needs it. This need for information could be modeled with Bell-La Padula model but would lead to problems: proliferation of security labels and categories, leading to issues because of the centralization of security clearances.

Even tight control of data may not always help as publicly traded companies need to release some data because of laws and they allow their partners access to some data. Taken together, this data that is innocuous by itself, will often allow deduction of confidential information. This is accepted and companies can still function correctly even if some of their secrets get out but not so when companies order lists and warehouse inventories get corrupted.

Enter integrity models: three most well known integrity models are Biba model [Bib75], Clark-Wilson model [CW87] and Lipner's integrity matrix model [Lip82] that is combination of Bell-La Padula and Biba models. Of these models, we will describe the first two.

3.2.3 Biba models

Whereas BLP guards confidentiality, Biba models [Bib75] protect integrity. There is actually a set of Biba models and term "Biba's model" usually refers to Biba's "strict integrity model".

In all Biba models, integrity and trustworthiness of data is denoted by integrity levels, denoted by $I(x)$, a concept similar but distinct to security levels. Security levels of objects denote their classification but object integrity levels ($I(o)$) depict how much trust there is on the information; the higher the integrity, the more confidence we have that a program will execute correctly and data is accurate and reliable. Subject integrity level ($I(s)$) describes how much we trust information generated by the subject: how trustworthy are her sources and how good the information she generates is.

Biba's Model (String integrity Policy)

Biba's model can be summed to three rules, as follows.

1. $s \in S$ can read $o \in O$ if and only if $I(s) \leq I(o)$
2. $s \in S$ can write to $o \in O$ if and only if $I(o) \leq I(s)$
3. $s_1 \in S$ can execute $s_2 \in S$ if and only if $I(s_2) \leq I(s_1)$

This is *the Biba's model*; it is the mathematical dual of BLP model. The model, shown in Figure 4(a) can be summed as "no read down, no write up". Biba's model does not allow change in subjects' or objects' integrity levels, making the model hard to use in practice. For practical usage, there are Biba models allowing such changes.

Low watermark policies

Low watermark policies allow change of integrity levels and either modification of objects at any integrity level or reading of objects at any integrity level.

Subject low watermark policy removes the first clause of strict integrity policy and allows subject to read any object o at any integrity level. Integrity level of the reading subject s is lowered to the lesser of $I(s)$ and $I(o)$.

This policy assumes that subject will rely on the data she has read from lower integrity level and therefore can no more be trusted to write on her earlier integrity level. Therefore her integrity level is lowered. It is shown in Figure 4(c).

Object low watermark policy states that subject s can alter an object o at any integrity level. New integrity level of o is the lesser of $I(s)$ and $I(o)$.

Object low watermark policy drives does not prevent improper modification, only makes the modification apparent by lowering the integrity level of modified information.

Both policies allow only *non-increasing* changes in integrity levels. This can eventually lead to situation where either all objects are at the smallest integrity level or to all subjects have clearance only to smallest integrity level.

Ring property

Writing to higher integrity levels can be allowed through *gates*: trusted procedures that perform consistency checks ensuring that objects in higher integrity levels do not get contaminated.

Subject s_1 can read objects at any integrity levels.

Subject s_1 can execute $s_2 \in S$ if and only if $I(s_1) \leq I(s_2)$.

This is named as *ring property* in [Gol05] but the name is easy to mix up with *ring policy* introduced in [Bib75] that does the reverse, allowing execution only if $I(s_2) \leq I(s_1)$. Ring property-solutions are employed in many computer systems where a user can change some data through a trusted interface, marked with *execute* in Figure 4(b). For example, in Unixes user cannot directly modify password file to change her password, but can change the password in the file through `passwd` program that is run with *supervisor* rights.

This usage of trusted procedures is developed further in Clark-Wilson model.

3.2.4 Clark-Wilson Model

Clark and Wilson argue [CW87] that both Bell-La Padula and Biba models are better suited for protecting confidentiality instead of integrity and therefore good for systems where confidentiality is the most important aspect, such as military systems. To protect the integrity of the data, the most important requirement in commercial systems, Clark and Wilson proposed a radically different model called Clark-Wilson model.

In Clark-Wilson model, data can only be manipulated through a specific set of programs; users have access only to these programs, never directly to data. This bring us to first difference between Clark-Wilson and Biba or BLP models: in Biba and BLP models, authorization allows subject to perform some generic action a, such as reading or writing whereas in Clark-Wilson model authorization gives permission to run a *well-formed transaction* through a specific procedure.

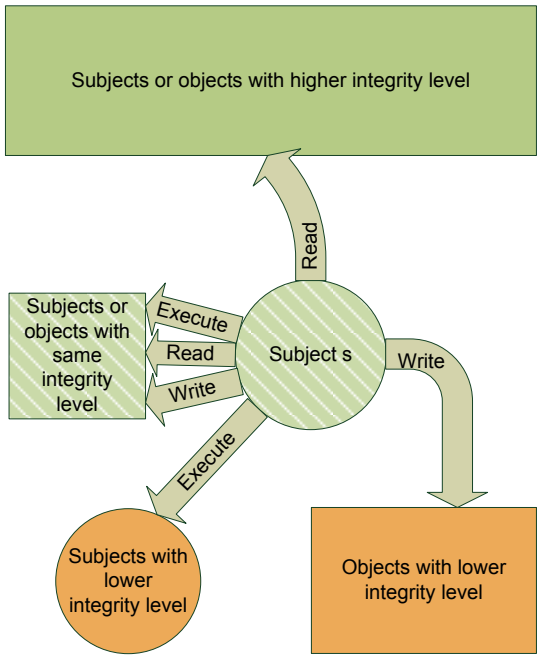
Clark-Wilson model divides data to two sets: to *constrained data items* (CDIs), which are governed by security model and to *unconstrained data items* (UDIs) that are inputs to the system from outside. Conversion from UDI to CDI is a critical part of the system and is managed by *transformation procedures* (TPs) that must be certified to be valid. In addition, *integrity verification procedures* (IVPs) ensure that the system is in a valid state each time the IVPs are run.

System security properties are defined through five *certification rules* and enforced by four *enforcement rules*.

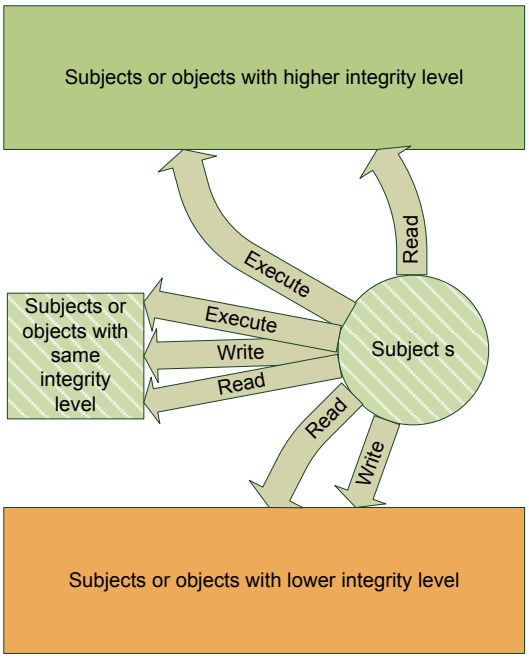
Certification rule 1: IVPs must ensure that *all* CDIs are in a valid state when any IVP is run.

Certification rule 2: TPs must be certified to be valid, that is they must transform CDIs in a valid state to another valid state.

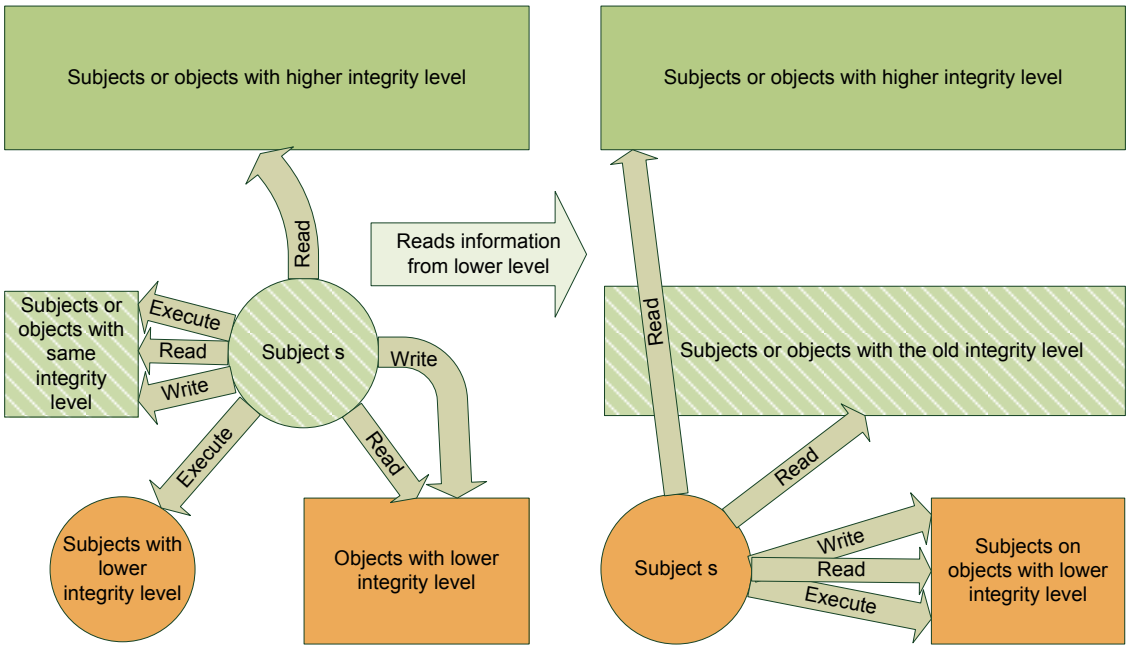
Each TP is certified only for some set of CDIs: a certified TP working with strings can corrupt the system when fed with pictures. This leads to first enforcement rule:



(a) Biba's model is static; its integrity levels do not change



(b) Biba's model with ring property. A subject can write to higher integrity levels but only through trusted procedures (execute).



(c) Subject low watermark policy. In low watermark policies integrity levels change based on used information. Here integrity level of the subject is lowered to the integrity level of the information used.

Enforcement rule 1: The system must maintain and protect list of CDIs that TPs are certified to access and manipulate.

Enforcement rule 2: The system must maintain and protect a list of tuples (*user*, *TP*, {CDIs}) that each user is allowed to execute.

This provides the basis of Clark-Wilson access control. Users can only deal with specific CDIs, using specific TPs.

Certification rule 3: The access rules must satisfy any separation of duty requirements.

Access rules must satisfy any separation of duty requirements that may be in place: if company has a policy that it requires two people to make an order greater than 10,000 $, access rules in Clark-Wilson model must also require it.

Enforcement rule 3: Each user must be authenticated on each TP execution attempt.

There is no need to authenticate users that log in the system from Clark-Wilson model's viewpoint as only UDIs can be manipulated without invoking a TP. Invoking a TP then requires authentication each time and the invocation is logged, giving certification rule 4:

Certification rule 4: Every invocation of TP must append enough information to reconstruct the operation to an append-only CDI.

Certification rule 5: Each TP that takes as input a UDI must transform it to a valid CDI or reject it and perform no transformation at all.

This means that all transformations from UDI to CDI are transactions: either transformation succeeds totally or system is restored to a state where no transformation was done at all, except for the log specified in certification rule 4.

Enforcement rule 4: Only a certifier of a TP may change the list of entities associated with the TP. Such certifier must never have "execute" permission on that TP.

The last enforcement rule presents separation of duties: same person who certifies a TP cannot be its user making this integrity enforcement mandatory instead of discretionary.

Compared to Bell-La Padula and Biba's model, Clark-Wilson model is more complex. It is clearly designed for commercial organizations with its emphasis on separation of duty and its control of integrity through verified TPs, both integral in prevention of insider fraud.

3.2.5 Role-based access control model

Role-based access control is a suitable model when access to data depends on user's *role*, such as job function. Sandha et al. describe a family of role-based access control models [SCFY96] of which here is described a model named $RBAC_1$.

Each user may have one or more roles, each having its own list of allowed actions and user as such does not have any other rights than the roles she belongs to. Roles themselves may contain other roles and thus allow composition of larger roles: a trainer is allowed to do anything a trainee can do as well as perform other actions. Thus, role-based access control is very suitable for security requirements of hierarchical organizations.

Role-based access control can also model separation of duty required by organizations and specify it centrally. Given actions a and b that need to be done by separate persons, restriction can be imposed that if $a \in allowed(A)$ and $b \in allowed(B)$, then for no subject s can belong to both roles A and B.

Although Role-based access control is a simple model and helps to reduce the complexity of managing access control, only a few operating systems, such as AS/400 and Windows 2000, support it.

3.2.6 Database privacy models

Some databases, such as census database, contain sensitive data that users are allowed to access in aggregate form such as "return number of males within ages of 40 to 49 living in Kansas". Yet multiple queries or queries that are sufficiently specific can be used to retrieve information on specific individuals: "return wealth of men living in Antarctica with age of 32 and blue eyes" will quite certainly return at most one result and if more results are returned, the query can be specified further.

To prevent this breach of confidentiality, Dobkin et al. [DJL79] showed that such systems need to take into account both query history and what can be inferred based on those older queries and current query when making access decisions. Denied queries, as well as denied accesses in other mechanisms can then be logged and actions taken if such queries are seen as intrusion attempts.

There have been other approaches after Dobkin et al. and for good background information and overview models, see Privacy-Preserving Data Mining [AY08].

3.2.7 Conclusion

Access control in security models comes in two versions: mandatory and discrete access control. All models described here are mandatory access models suitable to situations where access needs to be controlled in a top-down fashion. Bell-La Padula and Biba models are rooted in military-style security whereas Clark-Wilson and Role-based access control models were developed for business' requirements.

Operating systems rarely support mandatory access control models but applications used to share information within organizations often need to implement such a model.

3.3 Creating secure software

Research of secure systems has long been focused on creating secure operating systems and less so on other systems such as application programs. This is perhaps because creating a secure operating system is a much more daunting task than just a secure word processor. Security requirements differ: Operating systems usually cannot trust that all programs run on it will behave nicely but a word processor can expect that its user will behave well as the user will be the only one suffering from malfunctions.

Yet some software, such as software getting input from network (e.g. web server) faces same dilemmas as operating systems: intentionally misbehaving users trying to do something that they are not "supposed" to do. And if there are any vulnerabilities in the software, they may very well be able to do it.

This section discusses problems face in creating secure software, first discussing the problems, then solutions in form of development models and design principles.

3.3.1 Why developing secure software is hard?

Vulnerabilities in software come in two flavors: both a design *flaw* and an implementation *flaw* (usually called a bug) can create vulnerabilities. An example of design-level flaw is use of insecure cryptographic algorithm to provide confidentiality whereas implementation flaw could be that a secure cryptographic algorithm is implemented incorrectly allowing attacker to break confidentiality.

Creating working software is thought to be fundamentally hard [Bro87] and there are no techniques to eliminate or address all program flaws. It is then no wonder that information security, being implemented on computers, is hard. Gasser notes this and lists reasons why creating secure software is especially hard [Gas88].

- Security is fundamentally difficult. Creating large software systems is hard and such systems often are bug-ridden. Despite these bugs, systems are still usable as few bugs are fatal and others can be circumvented. Instead in supposedly secure software, each bug has potential to circumvent the controls in place and leave system open to attackers. As Gasser states, "You might be able to live in a house with a few holes in the walls, but you will not be able to keep burglars out."

- Security is an afterthought. As new customers are rarely won through security (barring limited sectors such as banking), but instead with appearance, usability, performance and number of new features. This often causes security

to be only an afterthought in software: when vulnerabilities are found, they are patched over instead of addressing the more fundamental problems causing vulnerabilities in the software.

- Security is an impediment. Goals of making software secure and making it easy to use often conflict. This is especially true when security is an afterthought instead of being built-in from the start.
- False solutions impede progress. False solutions give sense of security without actually solving the security problems.
- People are the problem, not computers. Gasser wrote the list 21 years ago and humans haven't changed much during that time. As stated in Section 2.3, people are often the weakest link in security allowing penetration of otherwise impregnable system using social engineering.

Creation of software can be thought to consist of three conceptually separate steps: specification, design and implementation. How the steps are interconnected depends on the development methodology and process used. Lest security becomes an afterthought, it must be considered in each of these steps. This thinking has lead many development methodologies to add special emphasis on security and to development of new processes, such as "The Security Development Lifecycle" [HL06] by Microsoft, instilled with security-awareness.

3.3.2 Development models for secure software

A fruitful discussion of different software development models could fill a book. Instead only two processes that were created for developing secure software are mentioned here. These development models try to give assurance that the system built is secure because it was built by a process that emphasizes security. Neither of the processes offers a "silver bullet" to secure development and instead just bring security out to normal software development process.

The security development lifecycle

"The Security Development Lifecycle" (SDL) by Microsoft is a heavy-weight process for developing secure software. Although adaptable to other types of software, it is geared towards producing shrink-wrapped software with long release cycles in a large company.

Before any software development is started, SDL mandates that each developer, tester and manager must complete a course describing basic concepts of computer security once a year. At the beginning of a development project, security requirements of the upcoming product are assessed and a person or team is charged with responsibility of managing and tracking the security of the product. Assessing security requirements means setting a minimum level of acceptable security that can

never be lowered, based on usage of the product (e.g. what personal information the product handles).

In design phase, SDL emphasizes following of "best practices", analyzing risks faced by the product and adding all security features required to the design. Best practices mean methods and tools that are seen as the most effective at the task at hand, for example, in design of control mechanisms, eight design principles described in Section 3.3.3. Risk analysis, or threat modeling as Microsoft calls it, enables security designs to better focus on the most important threats and reduces the possibility of overlooking important security features. Lastly, the design may need to be reviewed by an by an external reviewer if confidentiality of the personal information that the product handles warrants such a review.

Implementation phase again emphasizes following "best practices". Such best practices can be, for example, avoidance of C programming language functions that are error-prone or never to develop code that generates warnings from compilers or static analyzers. Special attention should also be paid to documentation so that users can use the product securely.

Before the product can be released, verification that the product works as intended needs to be done. Verification phase includes extensive testing and a "security push". Security push, described in detail in [HL03], is an effort to review and update documentation, code, design and risk analyses for changes that have occurred during the development.

Testing part consists of normal quality assurance testing reinforced with testing specifically for security vulnerabilities. One such testing method, promoted by SDL, is fuzz testing, a testing method where a program is input invalid or random data in hopes of finding bugs in input handling. This is especially important for programs written in C or C++ where such bugs can lead to vulnerabilities allowing, for example, remote execution of arbitrary code (see buffer overflows, Section 4.3.3). For a good overview of security testing problems, see [Tho03].

In release phase, there is still one final review of security and privacy. Risk analyses are verified to include all known threats and severity of known deferred bugs is re-examined. Responses to vulnerabilities found after release are planned. Then, finally, comes the release.

Secure web application development process

On the agile front, Ge et al. describe an agile process based on feature-driven development [GPP+06] that is perhaps more suitable than SDL for small companies and short release cycles. This process builds on existing feature-driven development (FDD) framework [PF02] by integrating security specific subtasks to it. Outline of the process is shown in Figure 4, where original parts of the FDD are marked by rounded rectangles. In FDD, the last two parts are done iteratively, repeating design and implementation for each planned feature. Security-enhanced version adds risk analysis to this iteration and decision of security policy to the start of the project.

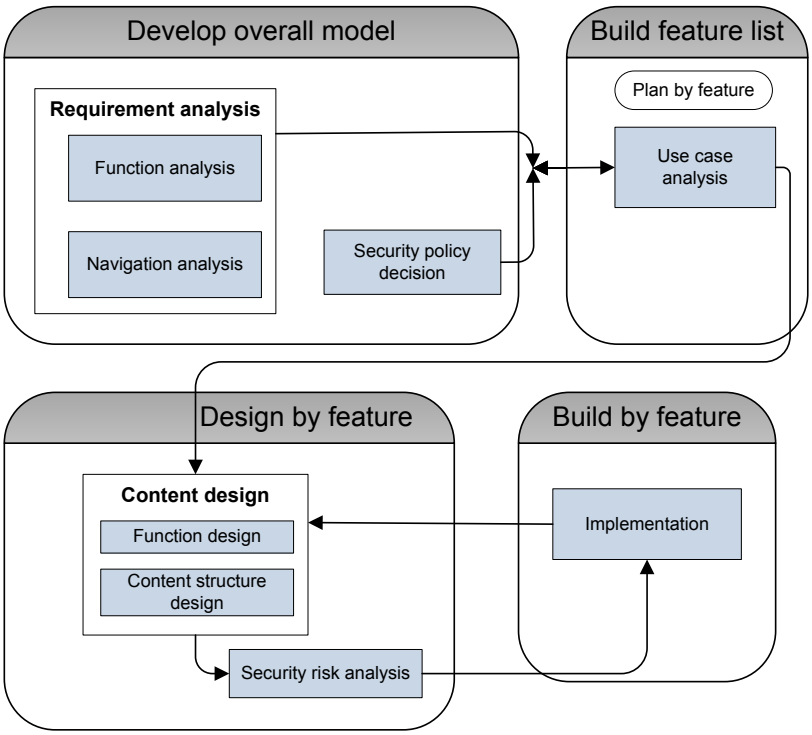


Figure 4: Overview of a secure web application development process. Adapted from [GPP+06].

Security policy is first set at the start of the project. Security policy is connected to the other requirements which means that when the requirements change, the policy needs to be reviewed and updated for possible changes. The other change to FDD is addition of risk analysis to design of each feature. Risks of a new feature are analyzed and proper responses decided as described in Section 2.2.4.

Other efforts for security in agile development seem also to focus on forcing developers to consider what security implications each feature they develop has. *Abuse cases* [SBK05] and *misuse cases* [Kon06] are examples of such development.

3.3.3 Eight principles of secure design

What ever the methodology used to develop software, there are certain principles that need to be adhered to in order to create secure software. These principles, first introduced in a seminal work by Saltzer and Schroeder [SS75] describe eight principles for design and implementation of protection mechanisms. It is good to notice that the principles are intended for the common situations where integrity and confidentiality are more important than availability.

The principles are as follows:

Economy of mechanism Simpler designs are easier to get correct. Simple design allows fewer ways to get either design or implementation wrong. Simple design

has fewer components and cases to be tested and thus more throughout testing can be done. This principle can easily become overlooked when security mechanisms are built incrementally or without proper thought.

Fail-safe defaults By default, all access should be denied. All access decisions should be based on permissions instead of exclusion. Lack of permissions caused by e.g. misconfiguration is easier to detect and safer than unintended access that would be allowed by exclusion based access decision. Another view of this is that should program be unable to complete its goal, the program should undo the changes it has already made to security state of the system. Then even a failing program will not compromise system's security.

Least Privilege Every subject in the system, be it program or user, should work with the least set of access rights needed for its job. If the subject temporarily needs more access rights, those rights should be relinquished *immediately* afterwards. This is to limit the damage that might result from implementation error, accident or deliberate attack.

Complete mediation Each and every access to every object must be checked for authority. Many contemporary operating systems check permissions only on the first access of object and cache the results for faster subsequent access even if the permissions change in the mean time. Combining both caching and complete mediation requires systematic updating of cached permission entries and leads to more complex system. This conflict with economy of mechanism needs to be inspected with healthy amount of skepticism.

Separation of privilege A security mechanism requiring two unrelated conditions to be met is more robust than a mechanism requiring only one. A single breach of security will then not be enough to compromise mechanism. This principle is equivalent to the separation of duty discussed in Section 3.2.4. Requiring a security token in addition to password to authenticate or needing to know both root password and to belong to a proper group in order to run programs as root user are examples of this mechanism.

Open design Security of the system must not depend on the secrecy of its design or implementation. Both these details can be worked out from disassemblies, black-box analyses or by non-technical approaches such as break-ins to implementing company.

Least common mechanism: Minimize the number of mechanisms that are shared among the users. Every shared mechanism represents a potential path to leak information between users. In desktop operating systems this is partially implemented: each process has its own virtual memory space but they use common file system, subject to access control.

Psychological acceptability As humans are often the weakest links in security, parts of security requiring human interaction should be as easy to do and use

as possible. Security related program should be intuitive to use, hard if not impossible to use in insecure manner and should give proper error messages. As not all concepts in security are simple enough to be explained in a few sentences and users rarely read error message, preventing incorrect usage may be the best solution. When creating informative error messages, no additional information should be divulged: if user entered wrong password when logging to a system, system should not tell user whether the password or user name was wrong. Revealing which (or both) were wrong, an attacker would know if such user name actually existed in the system.

3.4 Threat and Risk analysis tools

Paraphrasing [PP06], risk analysis is the process of analyzing a system and its deployment environment for threats and what potential harm they can do. There is some confusion in literature what actually risk analysis consists of as Microsoft uses term *threat modeling*[SS04] on what others call as risk analysis.

Before a risk analysis can be done, there need to be some threats to analyze.

Brainstorming threats without a process can cause some categories of threats being overlooked and thus more structured methods are needed. Three methods described next are tools to find threats and to estimate risks they give rise to.

3.4.1 Attack trees

Attack trees are a tool to describe and assess security of a system or a part of the system, introduced by Schneier [Sch99] and later formalized by Mauw and Oostdjik [MO05]. Attack trees aim to help understanding what goals attacks have, what attacks are likely to occur and what security assumptions underlie the system's security.

Attack trees are goal-oriented: a single attack tree has a root that is the goal of the attack, internal nodes that describe subgoals of the attack and leaf nodes that correspond to actions through which the (sub)goals are achieved. This orientation forces analyst to think like a attacker, to find a way some concrete goal such as "get company information" instead of "attack company web server".

If edges to child nodes are denoted with an edge as in Figure 5 then all the denoted subgoals must be fulfilled (and-nodes). Without the edge the child nodes are complementary: only one of the nodes needs to be true for the parent node to be true. The edges and nodes can also have additional information such as edges carrying the probability of the attack's success or carrying the cost of an attack to reach the subgoal (as in Figure 5). Adding this information is best done bottom-up: add costs and probabilities to leaf nodes and edges and propagate them towards the root.

Attack trees help to visualize the whole security perimeter and being easy to understand, attack trees communicate the threats well. Easy communication help

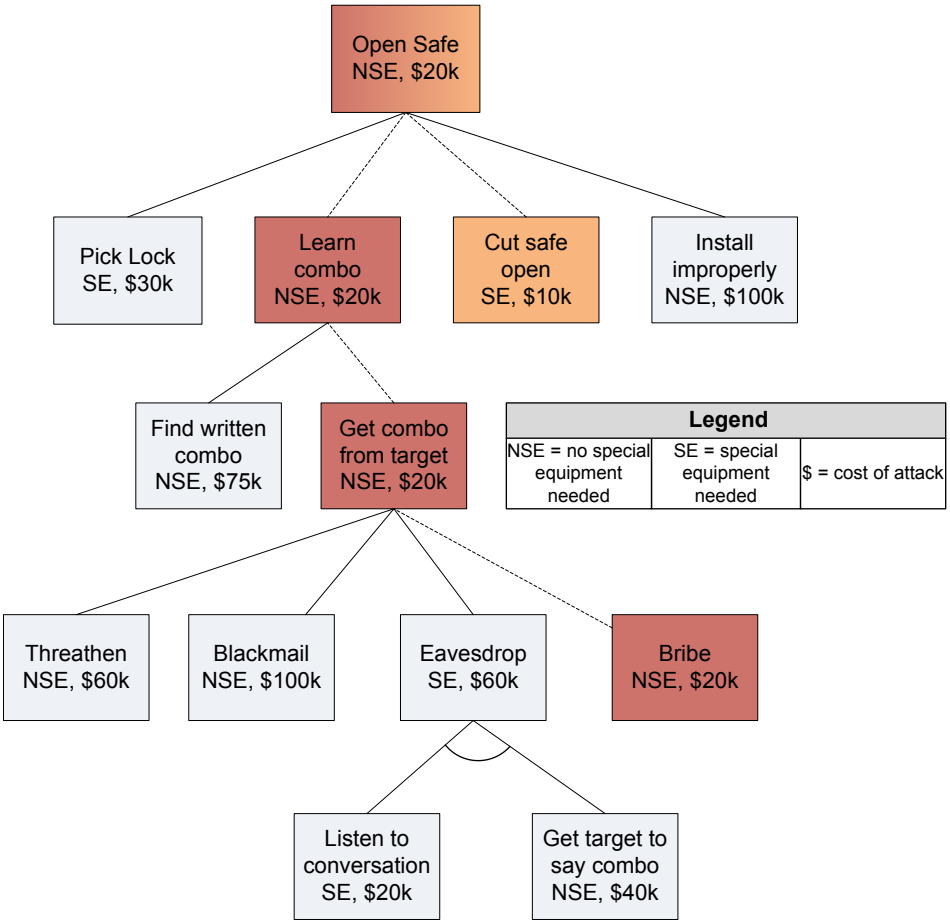


Figure 5: Attack tree for breaking a safe with costs and other requirements for subgoals. Red shows the attack requiring no special equipment and orange cheapest one. Adapted from [Sch99]

.

creation of good attack trees as the whole security assessment does not depend on one analyst for others can create subtrees of the attack tree. For example, a network security specialist can create an attack tree on attacking a system through network while a physical security specialist creates an attack tree describing different threats of physical penetration. These trees can then be combined to create a larger tree depicting system's security issues from physical and network view. This also works within same types of security issues as subgoals can be combined under same parent goal to form a wider attack tree.

All in all, creating attack trees is a good way to start the assessment of the system's security and when implemented using a computer software that does the cost-propagation automatically, quite easy to keep up-to-date.

3.4.2 Architectural risk analysis

McGraw describes architectural risk analysis as a necessity [McG06] as about half of the security problems stem from design flaws. Instead of ad-hoc analysis of architecture, McGraw describes an analysis process used by Cigital, a software security consultancy. The process takes a high-level view of the system, ignoring code-level problems, and tries to find out whether fundamentals of the system are in order.

First part of the process is to create a one-page overview of the system's architecture. Sometimes such an overview exists, but more often it needs to be developed through interviews and other data available of the architecture (partial documents, pictures, code).

After this is done, the actual analysis of architecture can begin. The analysis can be divided into three sub-processes:

1. Attack resistance analysis
2. Ambiguity analysis
3. Weakness analysis

Attack resistance analysis

First of the sub processes, Attack resistance analysis, is based on Microsoft's STRIDE approach [HL02] and pits known attacks, attack patterns and vulnerabilities against high-level view of the system. First, general flaws are searched using secure design literature and different attack types (as in 2.2.3) as checklists. Next, common methods to exploit software, called *attack patterns*, that are applicable to analyzed software need to be mapped.

This mapping can be done either using misuse or abuse case development [SO05, MF99] or using ready lists such as Common Weakness Enumeration [Mar07] that offer a taxonomy for vulnerabilities. Lastly, risks in the architecture need to be identified and evaluated, and known attacks understood. Attack resistance analysis is good for finding *known* problems but not for finding new types of attack.

Ambiguity analysis

Ambiguity analysis tries to find ambiguities in the way the system works or should work. In ambiguity analysis at least two persons separately ponder how the system or parts of it work, joining for a session to describe the system to each other analysis. Explicit explaining of the system's workings will bring up disagreements and misunderstandings which point to places of further analysis for weaknesses as hard-to-understand parts are hard to implement and use correctly. Ambiguity analysis, like code review that it strongly resembles, helps to pin-point parts of the code that could use *economy of mechanism, fail-safe defaults* and other design principles (Section 3.3.3) aiming for secure code. Ambiguity analysis may be the hardest part of the process to do right: McGraw notes that this sub process is best done with a team of very experienced analysts and best learned in an apprenticeship.

Weakness analysis

Weakness analysis sub process aims to understand the dependencies a piece of software has on external software and on the environment the software is run in. As most of the software is not written from ground up to a self-made microprocessor without an operating system, run-time environments, third party libraries, operating systems and environment all affect security of the software. As discussed in Section 3.3, some of these need to be trusted, but it is wise to consider at least what impact misplaced trust on third party libraries can have on the system. An important realization here is that it is not enough to just keep one's own code safe but that one needs also to track weaknesses in other software that is used, using, for example, security lists or computer security companies advisories.

All together, architectural risk analysis as described by McGraw is a step forward from ad-hoc approaches so common today. There is one problem, however. Description of Cigital's architectural risk analysis was frugal and seemed more like a marketing brochure as it created interest but did not specify process' implementation in enough detail. It probably is not meant as a marketing brochure and the opaqueness may be a sign that architectural risk analysis not being a fully specified process but is taught in an apprenticeship situation, as noted by McGraw.

Whereas architectural risk analysis took a whole program high-level view, attack surface measurement looks software from the point of data flow.

3.4.3 Attack surface measurement

It is common wisdom, codified in "economy of mechanism"-principle, that simpler systems are easier to secure. In the same way, the less of the system is exposed, the less there are places for possible vulnerabilities. Based on this thought, Howard et al. [HPW03] proposed the system's attack surface, the amount of system exposed to the world, as a metric of a system's security. This metric was further formalized and validated by Manadhata et al. [MTMW07, MKW08].

Exposition of attack surface measurement methodology here follows closely the ones given by Manadhata et al. First we need some definitions. Let S be the set of all systems, s be the system we are currently interested in and T be the set of all systems excluding s ($T = S \backslash \{s\}$). Additionally we have user U and data store D used by all systems in S. Tuple (U, D, T) is the environment of s called E_s (see Figure 6) and is the basis for rest of the definitions needed for attack surface measurement. Each of the systems in S has its own set of operations, each returning some output for and receiving arguments as input. These operations can be invoked by anyone in the environment. Additionally each system also has some set of communication *channels* that allow other systems and users to communicate with the system, such as TCP sockets, RPC facilities and named pipes.

Entry point is a operation, named m, used by system to receive data from system's environment. Entry point can either be *direct* or *indirect*.

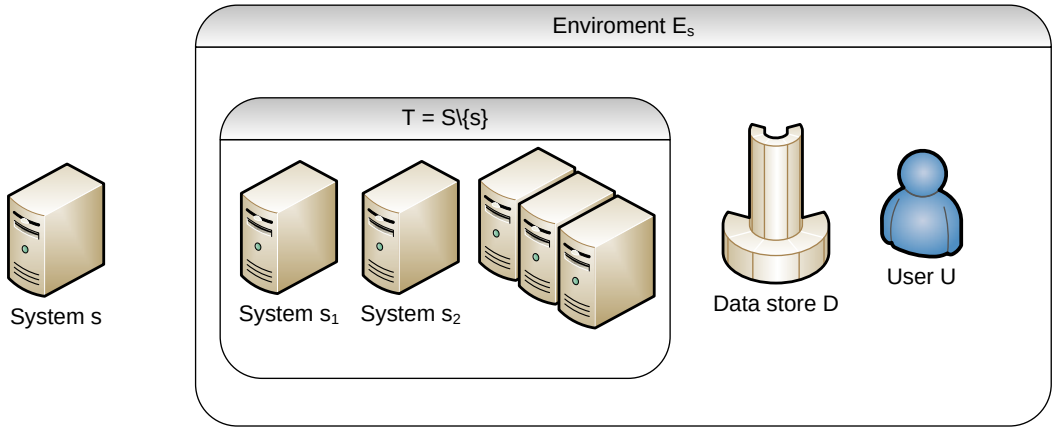


Figure 6: System s and its environment E_s

Operation m is a *direct entry point* if:

1. either user U or system s_1 invokes m and directly passes data to it.
2. or m reads data directly from data store D.
3. or m invokes operation $m_{outside}$ on system s_1 and receives data as a result from it.

Operation m is an *indirect entry point* if m receives data directly from operation m_1 that m_1 has either received directly from environment E_s or indirectly from another operation m_2.

For example, on a public web server, operation doing user authentication based on the user name and password a user has supplied is a direct entry point. System's logging operation that logs every login attempt could be indirect entry point if authentication operation would pass it the user name and password it has received from a user.

Exit point is a operation that sends data back to system's environment. Similarly to entry points, exit point can either be direct or indirect and the classification is quite the same.

Operation m is a *direct exit point* if:

1. either user U or system s_1 invokes m and receives data as a result from m.
2. m writes data directly to data store D.
3. m invokes operation $m_{outside}$ on system s_1 and passes data as parameters to it.

Operation m is an *indirect exit point* if m passes data directly to m_1 that m_1 either passes directly to environment E_s or passes to another operation m_2 that passes it indirectly to E_s.

As an example, a operation that writes to a log file is a *direct exit point* whereas a operation that gives current user name to a operation that creates a web page is an *indirect exit point*.

Untrusted data of a system s is data that s's direct entry point reads from D or s's direct exit point writes to D.

Attack surface of a system s consists of a set of entry and exit points M_s of s, of s's channels C_s and of untrusted data I_s that s uses. This set of resources that attackers can use either to send data to system or receive data from it is denoted with tuple (M_s, C_s, I_s).

Measuring Attack Surface

Simplest way of measuring a system's attack surface is counting the number of resources that constitute the attack surface. Yet it is clear that this straightforward operation overestimates the attack surface as it does not take into account differences between the resources, e.g. different operations may run on different privileges and different channels may have different amount of trust placed on them. This problem can be amended by estimating the contribution each resource has to the attack surface using *damage potential-effort ratio*. *Damage potential* is the amount of damage an attacker could cause to the system using the resource in an attack. How this damage is measured depends on the purpose of analysis: it can be technical impact (e.g. privilege escalation) or monetary loss. *Effort* measures the amount of work the attacker needs to do to acquire necessary access rights so that the resource can be used in an attack. Damage potential-effort ratio, *der*, estimates the resources contribution to the system's attack surface.

Summing over all there *der*-values gives us algorithm for attack surface measurement:

1. Given system s and its environment E_s, identify its attack surface (M_s, C_s, I_s).
2. Estimate damage potential-effort ratio, $der_m(m)$ for each operation, $der_c(c)$ for each channel and $der_d(d)$ for each datum.
3. Sum over different damage to potential-effort ratios for each resource type to get the measure of attack surface: $(\sum_{m \in M} der_m(m), \sum_{c \in C} der_c(c) \sum_{d \in I}, der_d(d))$.

Now only part left is the hardest part: estimating the damage potential and required effort. This part is highly subjective, but Manadhata et al. [MTMW07] give some advice for it:

Usually there is at least partial ordering between different *privileges* operations can be called, between different *protocols* channels can use and between different *types* of data. For privileges, for example, the operation may be called with *root* user, *normal*

user or *restricted* user privileges. Respectively a channel may have no restrictions on the traffic allowed (a raw TCP socket) or allow only one type of traffic (e.g. HTTP proxy). Additionally these resources can have different access rights, such as reading/writing sockets, and these all things need to be taken into account when estimating the damage potential and the required effort for each resource.

Manadhata et. al suggest a numeric value of $3 + k * \Delta_{current,lowest}$ for damage to potential-effort ratio based on the IMAP and FTP servers they studied [MTMW07]. On choosing k, Manadhata et. al suggest choosing values near 10 for privilege levels and between 10 and 20 for access rights. As choosing k can seem quite arbitrary and hard to estimate, it is a good idea to do sensitivity analysis for results. Manadhata et. al found that in cases where number of entry- and exit-points are similar between compared servers, emphasizing access rights or privileges using different k values can alter the results, but when the difference in entry- and exit-points between measured programs is "large", values of k do not matter.

Although attack surface is a metric for comparing security of different systems, it can also be used to the security of a single system. In such case, we will skip the summing step in measuring operation and instead just estimate the damage potential-effort ratios.

3.5 Conclusion

This section discussed both how to create secure systems and how to assess systems' security.

Important part of applications that allow multiple users to share data is access control. This section reviewed access control models emphasizing either confidentiality (Bell-La Padula) or integrity (Biba and Clark-Wilson) and introduced two models with other priorities (Role-based access control and a model for database privacy). Other way to group the models is to note that Biba models and Bell-La Padula model are useful in military or government usage as they inhibit flow of information. The other models are more useful in other organizations such as companies where both the integrity of the data and the ability to share it are important.

Building secure software requires more than just good access control. Section 3.3 reviewed problems of creating secure software, solutions in form of development models for secure software and in form of good design principles. As with other software development models, there is no single development model that fits everyone.

Finally, methods for assessing security of a program were introduced. Attack surface measurement and architectural risk analysis help finding weaknesses in the software. Attack trees allow composition of separate security sectors to a comprehensive model of whole system's security. The methods are usable together: attack tree is composed of weaknesses found in attack surface measurement and architectural risk analysis.

4 Web Applications: overview and security considerations

Nowadays, most companies and organizations offer their services in the Web, but this has not always been the case. Initially, all the interlinked pages in the Web were generated off-line, either manually or by a generator program. By 1995 need for dynamically generated web pages was clear and common gateway interface 1.1 (CGI/1.1) protocol was one answer to the problem.

CGI established a common protocol for interfacing to external software from the web server and allowed dynamic content: automatic generation of web pages and data contained within them. This can be considered as the birth of web applications as content of the pages was dynamic and could react to input from users and marked rise of web browsers as universal clients. While the content was dynamic, users' experience was less so: every interaction with the application, represented as a web page, required reloading of the whole page. This changed fast after a browser company named Netscape introduced client-side scripting language JavaScript in 1995.

Before JavaScript, web pages earning to be more dynamic had to resort to usage of browser plug-ins that had the downside of requiring a separate download and installation and had limited platform support, usually just Windows systems. Client-side scripting by JavaScript brought functionality and effects that had before only been possible through these plug-ins to all browsers that supported JavaScript.

JavaScript and later, XMLHttprequest (Section 4.2.3) allowed changes on web pages to happen without reloading the whole web page and brought web applications closer to normal desktop applications in usability and speed. This fluidity combined with dynamic scripting languages such as Perl and PHP and with template engines eased the creation of interactive applications and brought them closer to their desktop brethren in usability. Ease of developing such applications also lead to large amount of security problems in web applications that harmed the applications, their users, or both as many of the web application developers did not know the security issues of the web applications.

This leads us to this thesis' main theme, security of the web applications. Before tackling the case study (Section 5), we need to understand how web applications and vulnerabilities in them work which requires understanding the technology behind web applications. This chapter will first introduce HTTP, the protocol responsible for transferring data between a user and a website, and its secure brethren, HTTPS. After that, JavaScript and other technologies indispensable modern web applications, such as state management, are described. Finally, we present a high-level view of a typical modern web application and of typical security problems such an application contains.

4.1 Static pillars of the Web

The two most important technologies making up the Web are are HTTP with its secure sibling HTTPS, the protocols used to serve data in the Web, and HTML, the mark-up language used to describe the web pages a web browser shows.

HTML, abbreviation from hypertext markup language, is the most common markup language used to describe web pages. HTML document is a text document consisting of nested elements, marked by HTML tags such as `<HTML>` and this document is then rendered by the browser for graphical presentation. Basic understanding of HTML is presupposed here as it is not discussed here further except in context of DOM in Section 4.2.2.

4.1.1 Workhorse of the Web: Introduction to HTTP

The World-Wide Web and the Internet hardly need any introduction on this day and age. Still, a few words may be in order from the technical side. The World-Wide Web is a set of hypertext documents that are accessed through the Internet. Internet itself is a global network of computers interconnected using a standard Internet Protocol suite. For simplification, it can be said that every computer connected to the Internet needs to have at least one IP-address that is used only by it. Additionally, this computer may have one or more domain names, also called hostnames, that map an ASCII text to IP-address. These computers communicate with each other using protocols from Internet Protocol suite, such as HTTP.

The hypertext transfer protocol (HTTP) is, quoting Fielding et al. [FGM+99], "an application-level protocol for distributed, collaborative, hypermedia information systems." It is probably the most familiar protocol to casual users of the Internet, being the protocol used to serve web pages.

When Alice types *address* `www.example.com` to her web browser to direct it to load that page, she actually instructs the browser to make a *request* to `www.example.com` server. Server then responds to this request by sending a *response* that Alice's browser then renders to her screen.

Say, a user named Alice wants to visit *address* `www.example.com` with her web browser. She first types `www.example.com` to the address bar of her web browser (part 1 in Figure 7) and presses enter. Alice's browser, that is, her `User Agent` in more technical language, reads the address Alice has written on her browsers location bar and constructs an URI[2] from it. Based on this URI, Alice's browser checks what domain name is required and looks up IP address mapped to that domain name (part 2). Alice's browser then sends TCP packets containing the *request line* and *headers* (part 3) to the IP address looked up. When server has

[2] Uniform resource identifier is a string of characters that specify location and access of *resources* in the Internet. URI consists of two parts: scheme, a colon separator and scheme specific part. In our example, scheme would be "http" and scheme specific part "www.example.com:80/". For more thorough explanation, see [MDW02].

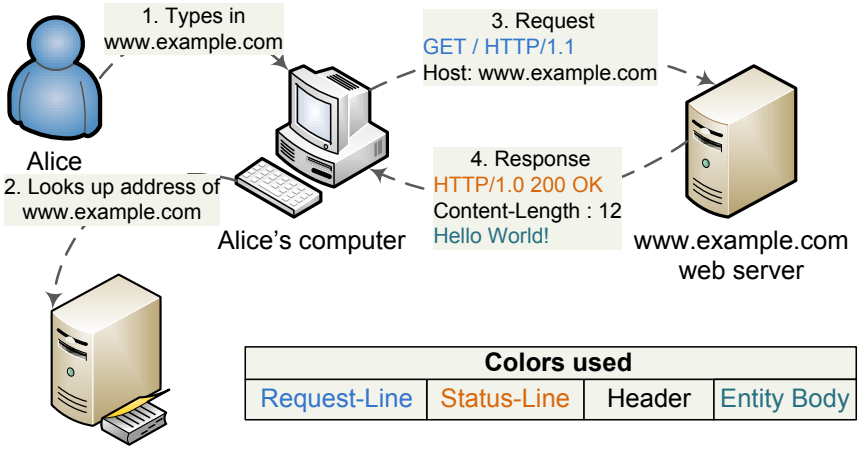


Figure 7: Alice types in www.example.com to her browser.

received the request, it reacts to it. In this case, Alice only has a simple "GET"-request where she only wants to get the root page ("/") of the server. Server reacts to this request by sending a response containing *Status-Line*, headers and possibly an *Entity Body* containing the actual page in *HTML* as in part 4 of Figure 7.

Depending on the response her browser receives, she may see a page rendered by her browser, an error message or some other response that can for example cause her browser request the same page from different address. After Alice has received her response to her request, this request is complete and server is allowed to forget about Alice.

Content of response and request messages is listed in Table 4. What a response or request message, such as what Alice sent and received, contains in its fullest is listed in Table 4. It should be noted that in the table, `<CRLF>` stands for carriage return character followed by a line feed character and it is used to separate each message line from others. Of the parts of a HTTP message, we are only interested in the request line, and in the request- and response-headers.

Request line

The request line lists the HTTP method used in the request. The latest specification of HTTP, HTTP 1.1 specified in RFC-2616 [FGM+99], defines 8 different HTTP methods, of which we are interested only in a few: *GET, HEAD* and *POST*. Of these, GET and HEAD are *idempotent*, meaning that $N > 0$ requests of those methods should have the same side-effect as a single request.

The *GET* method is the most common of the methods. The GET method tells the server to return the information identified by Request-URI in the responses entity-body. It should not result in any other significant action other than those required for the retrieval. As such behavior is not mandated by the RFC-2616, some web applications have used GET for other purposes. One such example is usage of GET

Part of the message	Examples
Request line specifies *method*, *Request-URI* and HTTP version of a **request**	GET /index.html HTTP/1.1<CRLF>
Status-line specific HTTP version, *Status-Code* and *Reason-Phrase* of a **response**	HTTP/1.1 200 OK<CRLF>
General-header fields specify headers usable both in requests and responses	Transfer-Encoding: utf-8<CRLF> Date: Tue, 15 Nov 1994 08:12:31 GMT <CRLF> ...
Request-header fields specify headers usable only in **requests**	Cookie: user=foo<CRLF> Host: www.example.com <CRLF>
Response-header fields specify headers usable only in **responses**	Set-Cookie: user=foo<CRLF> ETag: "b80f4-1b6-80bfd280"<CRLF>
Entity-header fields carry information about the entity-body or about the resource identified by the request.	Content-Length: 12<CRLF> Content-Encoding: text/plain <CRLF> ... ...
End of headers	<CRLF>
Entity-body contains the "payload" of the request or response.	<HTML><BODY>Hello!</BODY></HTML>

Table 4: HTTP Request or Response message

to delete data named in request-URI from the server, leading to problems if user-agent supposes that GET will just return data and for example pre-fetches such resources, causing deletion of all such resources named in a web page.

The *HEAD* method is like GET method except that server response must not contain message body. Response should be identical to a one from a GET request, save the missing message body, so that HEAD request can be used to get only the metadata of the page that is specified in the headers. A HEAD request can be, for example, used to check staleness of local cache or validity of a linked page.

The *POST* method submits data to the server to be placed under the resource identified by Request-URI. As an example, URI identifies a discussion thread on a message board and POSTed data becomes a new message within that thread. Data send to server is enclosed in entity-body and its length is signaled by "Content-Length" Entity-header. Although semantically meant to update resource pointed by URI, it is used as a generic method for uploading data to server.

These three HTTP methods cover most of the requests made in the Web and all that we will need for our web application. The other interesting parts, request- and

response-headers, being integral to state handling in web applications, are covered later in Section 4.2.4.

4.1.2 Confidentiality in the Web: HTTPS and PKI

HTTP sends all its messages, its requests and responses, in plain text. A request sent to `example.com` from Finland would pass through twenty or so different computers of which only few nearest computers could easily be checked for trustworthiness. All the computers in-between client and the server can arbitrarily read, modify, readdress and drop those requests and responses. This means that neither confidentiality nor integrity and especially not availability is assured when using HTTP. This is all fine as long as users and servers serving them don't have such expectations. For many uses, however, this situation is simply not acceptable. Enter HTTPS, the secure brethren of HTTP that fixes the confidentiality and integrity problems of HTTP. HTTPS provides confidentiality through *public-key cryptography*[3] and *public-key infrastructure*, and integrity through *message authentication codes* (MACs) [DA99, Res00].

Simplification of public-key cryptography can be stated as follows: it is a cryptographic approach where each participant has two keys, private key k_{priv} and public key k_{pub} with following properties

$$M = D(k_{priv}, E(k_{pub}, M)) \text{ and } M = D(k_{pub}, E(k_{priv}, M)),$$

where M is the message, E is the encryption and D the decryption function. With the assumption that it is, for all practical purposes, impossible to get k_{priv} from k_{pub} we have a system where a participant, be she named Alice, is the only one who can create messages that can be decrypted with her public key and only one able to read messages encrypted with her public key. Only problem left is how people wishing to communicate with Alice can get her *authentic* public-key as wrong key can lead to spoofing attacks where some third party purports to be Alice. This is called the key distribution problem.

HTTPS uses public-key cryptography similar to the one described and answers they key distribution problem with public-key infrastructure. In public-key infrastructure, there exists *certification authorities* that *certify* that a given key belongs to a given entity.

How it works in the Web, is shown in Figure 8. In the Web there are some top-level certification authorities (CA for short) whose *certificates* (called root certificates) are distributed with all the browsers. These certificates contain information such

[3] Cryptography is the art and science of secret writing. Historically cryptography has very much meant ways to secure messages' confidentiality, but nowadays it is also used to provide integrity and non-repudiation. As cryptography is a highly mathematical discipline and such treatment would not serve this thesis, only its usage is illustrated here. Readers more interested in cryptography should start with Schneier's *Applied Cryptography* [Sch96] and for more mathematical treatment, [Sti06, MvOV01].

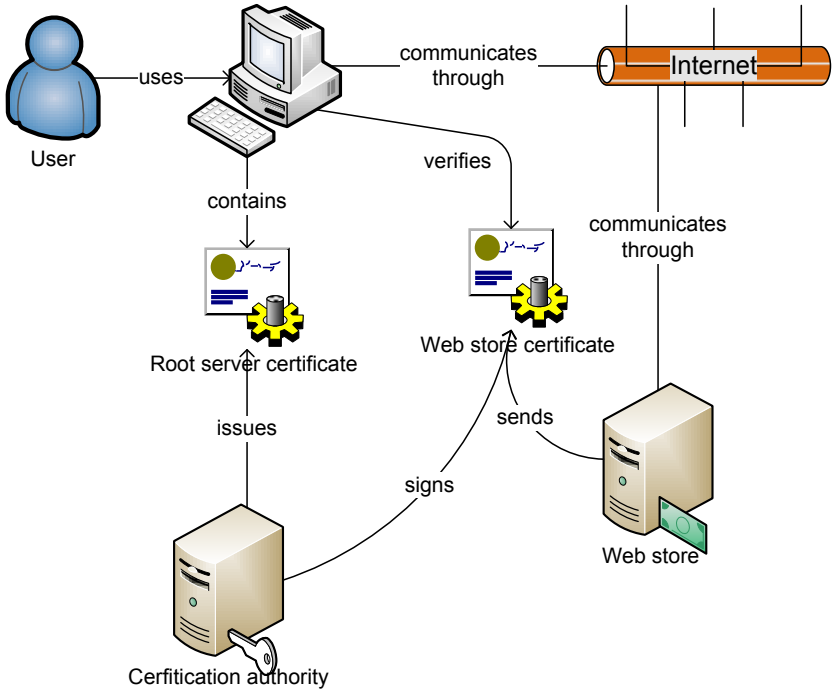


Figure 8: PKI as used in the Web

as the name of the CA, their public key and a *digital signature* of the certificate. These certification authorities then certify websites and other, lesser, certification authorities who then again certify other certification authorities or websites.

Now, when a user visits a web page using HTTPS, the server serving the page sends user its certificate. In this certificate, the digital signature is a digest of the certificate encrypted with CA's private key and thus decryptable with CA's public key. User's browser can thus easily verify that the server is trusted: if CA's public key decrypts the signature to a digest identical to what user's browser calculates from the rest of the certificate, CA vouches that the server is who it professes to be.

Usage of HTTPS as such is not enough to make web application secure. HTTPS just ensures that web application's security problems do not stem from clients talking to wrong servers or getting their messages read or modified, if the assumptions of the PKI hold.

These assumptions are two-fold: no one but the certification authority itself can digitally sign certificates with that authority's key and that certification authorities issue certificates certifying that website Y is X only when Y is X. Both of these assumptions have been shown to be false on some specific occasions: Stevents et al. [SSA+09] created certificates that seemed to be signed by certification authority and on some occasions, lesser certification authorities have created domain certificates by request of someone not owning the domain.

Despite these problems, HTTPS is usually the strongest link in web application

security. Credit for being the weakest link often goes to dynamic parts of the web application developed by the application developers themselves.

4.2 Dynamic web technologies.

Web application is an application that lives at least partially in user's browser. Web applications can either be developed to use only technologies native to browsers or to require plug-ins such as Flash or Java.

As early browsers were quite limited in their interactivity, developers that wanted to users of their website to have more fluid experience needed to use browser plug-ins. Using such plug-ins had the upside that instead of user submitting a text in a form and waiting for a new page to load, the users could use web applications with speed and interactivity rivaling those of applications running on users' desktop. The downside for web application developers was that that the users needed to have corresponding plug-in installed and enabled and the downside for user was that plug-ins often crashed the browser and had security problems and those again slowed the adoption of such plug-ins.

Nowadays such problems are much forgotten because plug-ins are more mature and, more importantly, because most of the web applications use technologies native to browsers. Browsers have caught up with plug-ins on what graphics and level of interactivity they can offer to users, mostly because of computers tens of times faster than during the early days of web and because of few key technologies such as JavaScript, DOM and Ajax that well study next. We will limit ourselves only to web applications utilizing such technologies.

Of the web browser technologies, none, save perhaps HTTP and HTML described earlier, are more important than JavaScript, the scripting language supported by all modern web browsers.

4.2.1 JavaScript

JavaScript is a scripting language developed at Netscape Communications and released 1995 with their Netscape Navigator client. It is the most important client-side programming language used in web, partly because it can easily modify web page DOM (See Section 4.2.2) and because it is the only one that is widely supported on all browsers. As a programming language, JavaScript can be categorized as a prototype-based multi-paradigm programming language that took its major ideas from Self and Scheme [wg]. Syntactically it is C-based but is otherwise much closer to languages such as Ruby, Scheme and Lua that have garbage-collection, first-class functions and other rich runtime features.

First important feature of JavaScript is its capability of *first-class functions*, meaning that JavaScript supports same operations on functions as it does on other objects: binding them to variables, creating and passing them as parameters and return

values in functions during runtime. These features are illustrated in listing 1.

Listing 1: Functions in JavaScript

```
function generator(i) {
    return function() {
        var tmp = i;
        i = i+1;
        return tmp;
    }
}

var from_one = generator(1);
var first_val = from_one(); //first_val is now 1
```

In the end of the code snippet, first_val is 1 but subsequent calls to from_one will return 2,3, ... as variable `i` is bound to the environment of unnamed function within `generator`-function. First-class functions allow idiomatic use of functional programming, a programming style that eschews side-effects to achieve modularity [Hug89].

Another defining feature of JavaScript is its *prototype-based* behavior reuse. Instead of inheritance based reuse common in many object-oriented programming languages, such as Java, Smalltalk and C++, reuse in JavaScript is realized by cloning existing objects. New objects in JavaScript are created either through constructor functions, such as cloning functions, or from nothingness using object literal syntax.

Listing 2: Creating new objects in JavaScript

```
/*Create object vehicle from nothing using
 *object literal syntax.*/
var vehicle = {name:"General vehicle"};

function Clone() { }
function clone(obj) {
    Clone.prototype = obj;
    return new Clone();
}

var car = clone(vehicle);
car.wheels = 4; //car now has property "wheels"
```

This is demonstrated in Listing 2 where first object `vehicle` is created out of nothing and then object `car` is cloned from it. Object *property* `name` in `car` is first looked up in `car` and if it does not exist in it, in `car's` *prototype* `vehicle`. If property "name" were not found there either, this prototype-chain would be walked through until either the property is found or some object's prototype is `undefined` in which case the searched property too is `undefined`. In this case, however, `name` is found in `vehicle` so `car.name` returns "General vehicle". If `name` property is changed in `vehicle`, it changes also in `car` as they refer to the same thing, *unless* `car's name` is set to point to something else first.

4.2.2 Document Object Model

If JavaScript could not manipulate web pages and their graphics, it would not be much of a use to web applications. Clearly this is not the case: JavaScript can manipulate the whole web page, all the graphics and text within it using DOM API. DOM stands for Document object model and is an object model that describes XML or HTML documents as a tree and allows programmatic manipulation of such documents using its application programming interface (API).

Listing 3: Sample HTML document

```
< HTML >< HEAD ></HEAD >
  < BODY >
    < A NAME = "#start">Start</A > of the document.
    < DIV ID = "unique">
      Some text here.
    </DIV >
  </BODY >
</HTML >
```

Consider a HTML-page in Listing 3. If we wanted to get HTML inside `<A>` anchor element, we can get it by reading DOM tree as shown in Listing 4. Should we want to change this text to something else, just setting `firstTag.firstChild.nodeValue` to something else would change the text in shown web page.

Listing 4: Reading text inside anchor element

```
var allTags = document.getElementsByTagName("A");
var firstTag = allTags[0];
var textValue = firstTag.firstChild.nodeValue;
```

Now, if only data could be send and read from web application server without reloading and re-posting pages, web applications using JavaScript to manipulate DOM could work as dynamically as normal desktop applications. Luckily, when developing Outlook Web Access 2000, Microsoft created a concept that does just that: a JavaScript object called XMLHttprequest [Hop07].

4.2.3 Ajax

Ajax stands for Asynchronous JavaScript and XML and is a group of related browser technologies that allow creation of web applications with interactivity close to that of desktop applications. The abbreviation was originally coined and defined in an essay by Jesse Garrett [Gar05] and brought two new technologies to web application development: XML as a data interchange format and XMLHttprequest as an asynchronous data retrieval method. XMLHttprequest is the more interesting of these two as XML is just a data container and currently has a lighter rival called JSON.

XMLHttprequest adds interactivity by allowing data transfer to happen asynchronously, without page loading. Figure 9 shows a life cycle of one such XMLHttprequest.

XMLHttprequest object is first created in JavaScript to send a request (usually GET or POST request) to the server and its `onreadystatechange` property is set to a callback function that we want to run after successful response. This request is then send by calling `send` property on the object and JavaScript interpreter moves onwards. Only when response to the request is received, does the JavaScript interpreter call XMLHttprequest object's `onreadystatechange` function which then can update JavaScript variables and DOM to display new results.

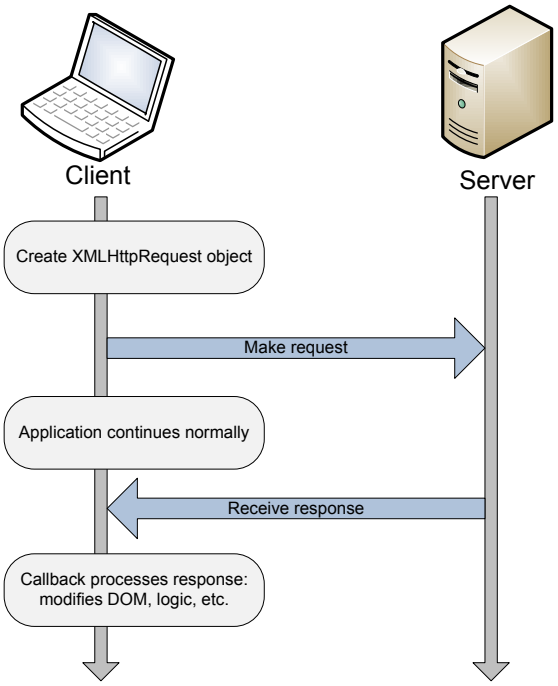


Figure 9: Life cycle of an XMLHttprequest

Last problem remaining is representation of the web applications state. As long as the application does anything more than just saves files or text that user sends to the server, the application needs to keep track of state. State of the application means recording what choices the user has made earlier to the extent they might be needed later in the application. As HTTP is designed to be a stateless protocol, state needs to be carried on some other level than protocol level.

4.2.4 State in Web applications

State in web applications means the same as in desktop applications: what modification the user has done, who the user is, what operation are we executing just now, etc. HTTP being stateless, it cannot be trusted to hold the state and instead web applications need to handle the state themselves. The are two possibilities to handle the state on the client or on the server.

The first possibility is implemented in web applications following representational state transfer style (REST) [Fie00]. In such applications, data of the application

is represented as HTTP resources and manipulated using different HTTP methods, for example resource `http://example.com/resources/cars/` denotes all cars and `http://example.com/resources/cars/1231` would denote a specific car. Then, a request with `GET` method would return a description of this specific car and request with `DELETE` method would remove the car.

The other choice is to have the server hold the state, stored in a data structure called *session*. The session-based methods are discussed more deeply as our case study employed sessions (Section 5.3.1). As each user of the application has her own session, these sessions need to be mapped to users. This mapping is done by a session identifier, a string that is unique to each user having a session on the server. This session identifier can be stored in three different ways: using URI encoding, hidden forms or, most importantly, cookies.

Session encoding in URI or hidden forms

State encoding in URI or hidden forms is nowadays mostly used only when cookies cannot be used for some reason. In URI encoding, the data is appended in the resource part of an URI (`http://example.com/re/sour/ce`) as this value is passed back and forth between client and server. In hidden forms technique, session information is carried in each of the web application's pages in a form element hidden from view. Both have some downsides: URI encoding makes sharing URIs hard and staying logged in a service is hard in both. If session identifier is encoded in URI, then Alice has to manually sanitize URI she sends to her Carol, otherwise Carol can act as Alice in the application; Alice needs to manually remove the part identifying her session from link before she can send it Carol (e.g. (`?sid`) from `http://example.com/resource?sid=aEdNcAMqDjqA`).

Because of these downsides, these techniques are usually only used when cookies cannot be used.

Session encoding in Cookies

Cookies have neither of the above-mentioned problems. A cookie is a string of text send back and forth between the server and client. First version of cookies was informally specified by Netscape Communications in their Mosaic Netscape 0.9 beta and more formally in RFC 2109 [KM97] which was later theoretically made obsolete by RFC 2965 [KM00]. In practice, only few browsers and websites support RFC 2965 cookie mechanism. Cookies are transferred in HTTP headers: a cookie is set by server using `Set-Cookie` response header and sent back to server in `Cookie` request header.

Before more details are exposed, an example of cookie use in web store is shown to illustrate the concepts. Here only headers that are integral to the example are shown.

1. User logs in by filling user name and password form fields and clicking "login"-button. User Agent sends the corresponding request.

   ```
   POST /store/login HTTP/1.1
   [Entity-body containing form data]
   ```

   Here Entity-body contains data from both the user name and password fields.

2. Server acknowledges user's logging in and responds to User Agent by settings a cookie to reflect user's identity.

   ```
   HTTP/1.1 200 OK
   Set-Cookie: User="Alfred_J_Kwak"; Path="/store"
   ```

   Here server restricts the cookie only to paths with prefix `/store`. The cookie could also be set using JavaScript. In that case the sent response would contain an Entity-body with following JavaScript snippet:

   ```
   document.cookie = 'User = "Alfred_J_Kwak"';
   document.cookie = 'Path = "/store"';
   ```

3. User Agent adds an item to shopping basket.

   ```
   GET /store/additem?id=3 HTTP/1.1
   Cookie: User="Alfred_J_Kwak"; $Path="/store"
   ```

   The old cookie is sent to server as `/store` is prefix of `/store/pickitem?id=3`.

4. Server then acknowledges the action and adds selected item to session cookie

   ```
   HTTP/1.1 200 OK
   Set-Cookie: Id1=3:Qu1=1:User=Alfred_J_Kwak; Path="/store"
   ```

   Here `It1` and `Qu1` mean the identifier and quantity of the first item in the shopping basket.

5. User decides that it is time to order, selects a shipping method from menu and clicks "Order" button.

   ```
   POST /store/checkout HTTP/1.1
   Cookie: Id1=3:Qu1=1:User=Alfred_J_Kwak; $Path="/store"
   [Form data specifying UPS as the courier]
   ```

6. Server reads name of the courier from Entity-body and user's address from its database. It then adds a job to another service to send selected items to chosen address using UPS. As items in the shopping basket are already bought, there is no further use for them and they are discarded.

```
HTTP/1.1 200 OK
Set-Cookie: User="Alfred_J_Kwak"; Path="/store"
```

User is still logged in and able to do further purchases but this transaction is complete.

In the example, cookies are used to store user's identity and what products the user has selected from the web store. These values are saved in one or more comma-separated `name:value` pairs that need to be kept confidential. This confidentiality is enforced by variety of cookie access rules.

Without any qualifiers, browser sends cookies only if hostname part of the page URI matches stored hostname of the cookie. For example: a cookie set by `http://example.com` is send to page `example.com/~user` but not to `www.example.com` even if they are the same server. As such a coarse-grained policy can cause both confidentiality and usability problems, scope of the cookies can be controlled in a more fine-grained way by three attributes: `Path`, `Domain` and `Secure`.

`Path` attribute can be used to restrict the allowed paths only to paths that have the same prefix as given attribute value. So Alice (`http://example.com/~alice`) could prevent Mallory (`http://example.com/~mallory`) from reading her cookies by appending `Path="/~alice"` to her cookie.

`Domain` attribute allows different sub-domains to access the same cookie: normally `store.example.com` cannot read login cookie set by `login.example.com` but appending "`Domain=".example.com"`" to `Set-Cookie` header allows different sub-domains of `example.com` to read the cookie. If this value does not start with a dot, dot is prepended to the value set. Now if domain value starts with a dot (it is set explicitly) and is of form .a.b where a.b is not a top-level domain, this cookie is send to all websites with hostname of form c.a.b, d.c.a.b, etc., but *not* to websites of form hostname a.b. Hostname a.b is only matched with implicit domain in a cookie send by a.b.

Lastly, `Secure` attribute denotes that cookie should only be sent back to server using at least as secure channel as was used to receive the cookie. In practice this means that cookies that are set using HTTPS connection with `Secure` attribute are not sent over normal HTTP connection.

Being used to save confidential information such as user credentials or session identifiers, cookies have been common attack targets. These attacks have usually been tied to incorrect cookie settings such as `Secure` attribute not being set or to web applications JavaScript handling vulnerabilities, such as Cross-Site Scripting (Section 4.3.5).

Now with being able to handle applications state, all ingredients needed for web applications have been introduced.

4.3 Web applications

4.3.1 Common architecture

Conceptually, web applications can be thought to consist of three tiers:

1. *Persistence* tier serves data and saves the changes users have done to the data.
2. *Processing* tier handles the information processing that usually is application's main task.
3. *Interface* tier displays application's user interface: it visualizes the data sent to it by processing layer and transmits users' commands back to processing layer.

Where these tiers actually reside in the web application varies from one extreme to another: from an implementation where the client loads a JavaScript from server and this script contains the whole application to an implementation where user uploads a file to the server and is redirected to another page containing the results. In the first extreme, all of the application is run on the client end and in latter one, on the server end.

As interesting security issues appear only when the client is not handling all the data, we will focus only on such cases. Setup of such a web application running in the Internet is shown in Figure 10. An important part of such applications is retrieving

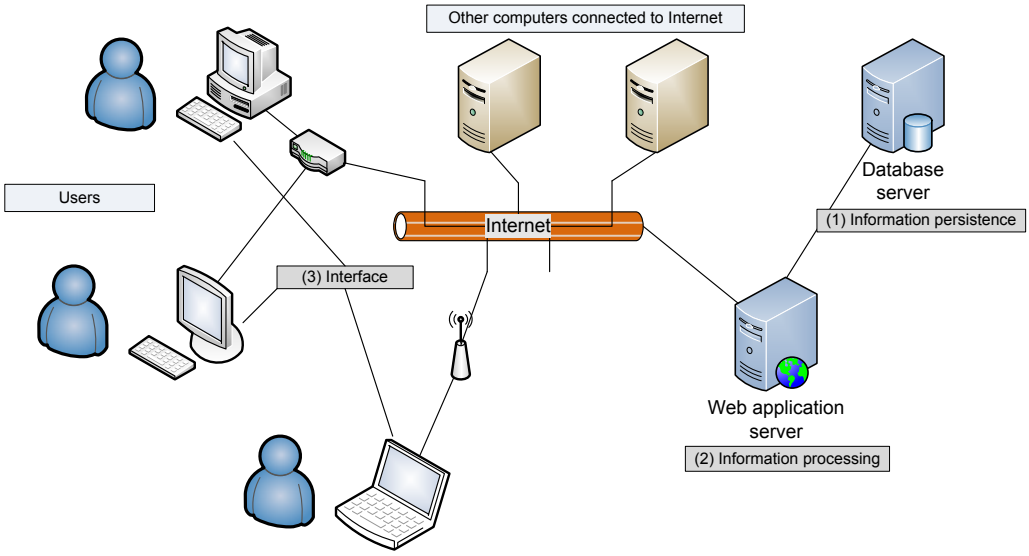


Figure 10: Deployment of a typical web application in the Internet

data from the database and displaying it in some form to the users in their interface. This is done through web pages and the web pages need to be generated at least partially based on the user and data in the database (Alice needs to see a different

welcome page than Carol). Most common way to generate these web pages is to use a *template engine*.

As shown in Figure 11, template engine takes a *template* for the page user requested and generates a HTML page based on information in the database and in the session. The templates template engine uses are a mixture of HTML and *template language*, latter being responsible for all the dynamic content.

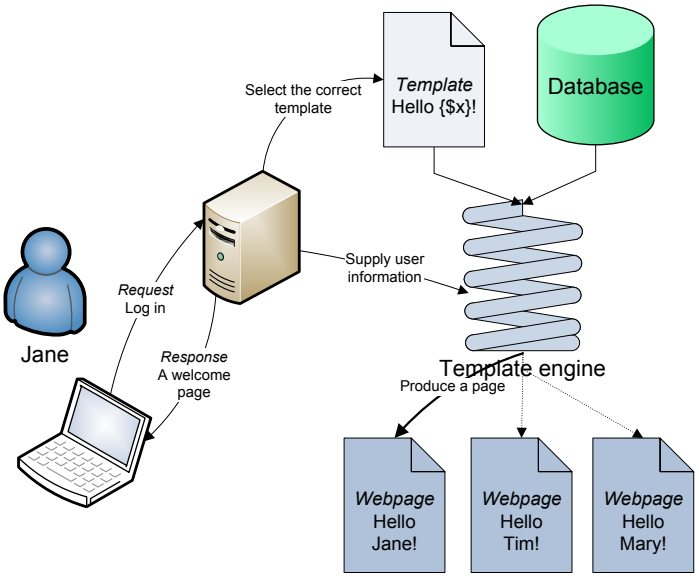


Figure 11: Template engine generates web pages dynamically from database data.

Template engines are not required for creation of dynamic web applications for all HTML could be generated programmatically. This approach has logic and presentation in the same place, whereas usage of template language allows interface designers to create interface separately using HTML and template language and developers to focus on creation of the actual application logic elsewhere.

In web applications, all data served from the server is allowed to do same things: to read cookies specific to the website serving the application, make actions behalf of the user, etc. To be able to use such web applications, users choose to trust the server but what they do not choose to trust is the other users of the application. All web applications need to ascertain is that this division between trusted and untrusted does not get intermingled, for it could lead to attacks such as XSS (See 4.3.5).

4.3.2 Common security problems

Arising from the same technologies used, different web applications have shared the same vulnerabilities. Although these vulnerabilities are not the only ones found in web applications, they still represent the lowest hanging fruits for attackers and therefore warrant special attention. A few of these low hanging fruits are introduced

here and more can be found in top lists by OWASP [vdSWW07] and CWE [Chr09].

4.3.3 Buffer overflow

Buffer overflow is the old champion the of CWE yearly vulnerability charts: it held the first spot for years before web application specific vulnerabilities took over. Buffer overflow bugs have made possible many Internet worms such as Morris Worm and SQL Slammer that both had great a effect on the Internet. Buffer overflows can exist in programs created at least partly using languages that are not memory-safe. There is a large variety of different types of buffer overflows but the most archetypal one is the stack-based buffer overflow bug seen in Listing 5.

Listing 5: stack-based buffer overflow

```
void o_flow(char *input) {
    char buf[256];
    char *start = buf;
    while(*input){*start++ = *input++;}
    /* Do something with buf */
}
```

Buffer overflows vulnerabilities can either lead to data-corruption and crashing of the process of thread running the code or to remote execution vulnerability. The latter is clearly more significant problem as it can allow the attacker to take over the whole computer running the web application meaning loss of confidentiality, integrity and availability while the former just causes loss of availability.

The problem with buffer overflows is that even though a memory-safe language might be used, its runtime and some, particularly I/O- and computationally heavy libraries, are most probably programmed in a memory-unsafe language such as C/C++.

Buffer overflows are best avoided by using constructs that prevent them: more sophisticated data structures than arrays or memory-safe languages. Barring these possibilities, memory debuggers and good engineering guidelines are next choices.

4.3.4 SQL injection

SQL injection is another case where input from users is treated too gullibly. In this case, the persistence tier is attacked as queries to databases are created dynamically based on user input.

Listing 6: SQL injection

```
conn = pool.getConnection( );
String sql = "select * from user where username='" +
    username +"' AND password='" + password + "'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
```

Problem here is that `username` and `password` are based on user input and that the creation of SQL statement is handled as a string concatenation without escaping the concatenated parts to a form suitable for SQL. Depending how SQL query results are handled, this can lead to various damages:

In authentication: setting username to "`root --`" reduces query to "`select * from user where username='root'`" and thus allows user to log in as a root user. The trick is that "`--`" is SQL token meaning that rest of the line is comments only and should be ignored for query purposes.

In output: setting both username and password to "`1' OR '1' = '1`" would reduce query to "`select * from user where username='1' OR '1' = '1' AND password='1' OR '1' = '1'`". Here the first two conditionals, "`username='1'`" and "`'1' = '1' AND password='1'`" rarely hold but the last conditional "`'1' = '1'`" is always true, causing whole "user" table to be printed instead of one user and corresponding password.

A good way to avoid SQL injection is to add an abstraction layer between SQL statements and strings. This creates a single point that can take care of escaping input strings and of prohibiting unwanted sub queries.

4.3.5 Cross-site scripting

Cross-site scripting [Rit07] is even more common than SQL injection and has lately topped both the CVE and OWASP vulnerability charts. It is usually abbreviated as XSS instead of CSS which is used by another web technology, cascading style sheets.

In short, XSS means that an attacker gets JavaScript they wrote served to users of some site which allows them usually do everything that those users could do. This includes stealing personal information or changing the content of the page served.

XSS comes in three varieties: local [Kle05], reflected, and stored. In all these attacks, web application displays JavaScript or HTML content from attacker's input. In stored attack attacker has successfully stored this data to web applications database and in local or reflected attack this data comes from current request. For example, suppose we have a following page in our JavaServer Pages web application:

Listing 7: JSP-page vulnerable to reflected cross-site scripting

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html><head><title>XSS page!</title></head>
  <body>
<%  String msg = request.getParameter("msg");
    out.write("Hello " + msg);%>
  </body>
</html>
```

This piece of code is vulnerable to reflected XSS attack: suppose a user is tricked to go to URI of `http://example.com/?msg=%3Cscript%3Ealert%28%22test!%22%29%3B%3C/script%3E`. Just visiting this address would cause a JavaScript box saying "test!" to appear as URI-escaped characters are unescaped to form string `<script>alert("test!");</script>`. This may not seem dangerous but instead of showing a greeting, the supplied JavaScript could have sent user's session cookie to some other address, allowing session hijacking or had the website belonged to a newspaper, added a story about a company posting giant losses and thus manipulated stock market.

Like SQL injection, XSS is fundamentally about output encoding: how user-supplied data is written to database query in the case of SQL injection or in XSS case, how user-supplied data is rendered to a web page, be the data from database or from a request. A naive approach would be blacklisting of HTML elements and JavaScript and better working solution is to create a white list of allowed HTML elements or to require users to use a completely different markup language than HTML.

4.3.6 Cross-site request forgery

In Cross-site request forgery (CSRF or rarely XSRF) a user is logged in to a web application and user's user agent has a cookie to prove that. When this user visits another website that site can make request on behalf of the user by JavaScript or bare HTML tags. Let us assume that Alice is logged in to her banking web application, located at `http://bank.example.com`. Now if this web application were vulnerable to cross-site request forgeries, Mallory could craft a page shown in Listing 8 that would make deposit from Alice's account to Mallory's.

Listing 8: Different forged cross-site requests

```
<html><head><title>Page creating CSRFs!</title></head>
<body>
<!-- Using GET -->
<img src="http://bank.example.com/deposit?from=alice
          &to=mallory&amount=10000">
<!-- Using POST -->
<script>
var xmlhttp=new XMLHttpRequest();
xmlhttp.open("POST", 'http://bank.example.com/deposit', true);
xmlhttp.send('from=alice&to=mallory&amount=10000');
</script>
</body>
</html>
```

This works for one simple reason: every time a request is done to a website that browser still has valid cookie for, that cookie is send with the request. website receiving the request has no way on knowing whether user intended the request or whether it was initiated automatically by user's browser because of some other web site. This is a case of *confused deputy* problem (Section 3.1.3): requests originating

from the web pages a user visits are made with the same level of authority as the user's explicit requests because the browser always sends the user's cookie to the website.

Where XSS exploits the trust users have on the web application, CSRF exploits the trust the web application has on the user. Therefore web application needs to be sure that requests are made by the user.

Adding an unforgeable token to each web page that then needs to be submitted back to server with each request is an adequate control to prevent CSRF. This unforgeable token might be a large random number from cryptographically secure random number generator or the user's cookie, both can be embedded within the requests to prove authority.

Yet these controls are not enough to prevent another closely related problem discovered in 2008, clickjacking.

4.3.7 Clickjacking

The main idea of clickjacking is to fool user into clicking buttons resulting different actions than what user expects. Consider a case where Alice is authorized to edit users' access rights in a certain web application, called "foo", and the steps required for edit are:

1. Click on the user to edit.
2. Click what rights to give the user, read, all or admin.
3. Click "save" button.

Now Mallory sets up a web page that contains this "foo" in an `iframe` meaning that page from "foo" is served within Mallory's page. Mallory can then arrange her page so that it covers page from "foo" and creates fake buttons for Alice to click. Clicks from these fake buttons are then submitted to "foo" and with properly set up page, Mallory can trick Alice to give her admin rights.

Protecting against clickjacking is best done by ensuring that the web application cannot be included in an `iframe`. Shortest way to do this is to include JavaScript snippet from Listing 9 to all pages web application serves.

Listing 9: JavaScript for breaking out of frames

```
<script>
  if (top!=self) {
     top.location.href=self.location.href
  }
</script>
```

This solution will not work if JavaScript setters, the functions that are called when some JavaScript objects value is set, are redefined as `top.location.href=<value>`

may not do what is intended. A more reliable way is to add an `onclick` function to all web application's buttons that is run every time a button is clicked and denies the request made by the click if the button is within an `iframe`.

As these both are a bit cumbersome and do not work when JavaScript is disabled, browsers may need to come up with some protections of their own to solve the problem.

4.3.8 Additional information

More information on these and other vulnerabilities can be found in Common Weakness Enumeration (CWE), a strategic software assurance initiative sponsored by U.S. Department of Homeland Security and run by MITRE Corporation [Mar07]. It classifies and categorizes vulnerabilities in software. CWE site has these categorizations available for everyone to use with details explaining exploiting potential, examples of the weakness, potential mitigations, and applicable platforms among others. CWE also lists observed examples (actual vulnerabilities found in software) and links to corresponding CVE (Common Vulnerabilities and Exposures) listing giving more info on that specific vulnerability.

4.4 Conclusion

This section described technologies used to build modern web applications. HTTP and HTTPS constitute the transport layer used to transfer data of web applications. HTML, JavaScript, DOM and XMLHttprequest are the tools for building such applications. As the tools used to built the applications are common to all web applications, the web applications often share same types of vulnerabilities.

Armed with knowledge of information security, security assessment methods and web applications, we are ready to launch our case study. This case study will assess and enhance security of a one mature web application.

Part II
Efecte - a case study

5 Analysis of the Current Efecte System

Now that both information security with all its concepts and models and web technologies are all well understood it is time to put them to good use. This section will introduce object of the case study, a mature web application called Efecte. Such a mature application provides excellent grounds for assessing security but implementing new security feature will be harder than earlier in the development.

After a short introduction to Efecte as a company and a product, an overview of Efecte web application is presented. Latter half of this section assesses Efecte's security.

5.1 Overview of Efecte as a product and a company.

Efecte is an enterprise resource planning (ERP) solution for IT-departments developed and marketed by a Finnish company Efecte Corporation. As both company and the product have the same name, using just Efecte without qualifiers will from now on refer to Efecte the product. Efecte is implemented using Java and Java-specific web technologies such as Servlets and JavaServer pages (JSP).

At its core, Efecte uses dynamic object model [RTJ00] that makes it feasible to quickly adapt Efecte to customers' needs and requirements of modeling their IT domain. This modeling can be done by domain experts without the need of software developers. This allows the developers to do their normal development instead of customization.

Without digressing further to the dynamic object model, its upsides include fast and easy modeling of domains but it has a few major downsides: it is quite rare pattern but very pervasive in applications using it and it carries a performance penalty because of additional indirection.

This pervasiveness is worse from developers perspective as it hinders the use of many libraries that use Java's normal object model in their modeling. Additionally, as this pattern is quite rare, there are not many libraries, especially security ones, supporting such modeling directly.

As a consequence, Efecte developers have had to develop many of the security features by themselves. Such features are not as widely used and tested as the ones provided by more commonly used libraries. Thus this work to assess and improve Efecte's current security.

So far, security work done on Efecte has been quite ad-hoc, based on problems found by developers themselves or in black box security analyses. Efecte has been subjected

to such analyses multiple times. These analyses have been quite superficial and mostly brought up weaknesses and vulnerabilities that were known in advance. This superficiality of black box testing comes as no surprise as no analysis on architecture and comprehensive security solutions can be made. This work will continue onwards from those black box testings and do a more comprehensive analysis on Efecte with the help of access to both source-code and architecture descriptions. Especially the latter is important as architectural weaknesses are responsible for half of the vulnerabilities according to McGraw[McG06].

Before a proper analysis can be done, Efecte's current state and architecture need charting and introduction.

5.1.1 The Efecte data model

Efecte is mostly used to model computers, networks and the software connected to those and yet the data model underneath knows nothing of such concepts. As stated earlier, Efecte uses an dynamic object model close to the one described by Riehle et al. [RTJ00] where key concepts are *data cards, templates, classes, attributes, meta-data* and *handlers*. Compared to static object-oriented model used in Java or in C++, data cards correspond to instances of different classes, known as templates in Efecte and those classes contain instance variables that corresponding to Efecte's attributes. Efecte classes do not map directly to Java or C++ concepts and the closest thing matching may be traits [BDNW07] that originally only brought behavior, that is, methods, but have later been extended to contain also instance variables.

As seen in Figure 12, one or more attributes are grouped together to form a class; one or more classes are grouped together to form a template and those templates are then instantiated to data cards.

So far this may seem as a cumbersome way to model a static IT domain. Part of this intricacy comes from the fact that although Efecte is marketed as an ERP solution for IT, it is used in other domains such as contract management and the modeling tools need to be usable in those domains as well.

The other part comes from the fact that Efecte only springs to life with usage of *Handlers* and *Listeners*. Handlers specify actions that are taken when attributes are modified and listeners specify actions that are taken when the whole data card is saved. These two are then used by domain experts to create *business rules*[Ros03], rules that specify constraints, derivations and facts concerning entities that are significant enough for customer to be modeled in Efecte.

Setting up these rules is tasked to domain experts who model the customer's system at the deployment phase. This setup is done in Efecte using a web browser.

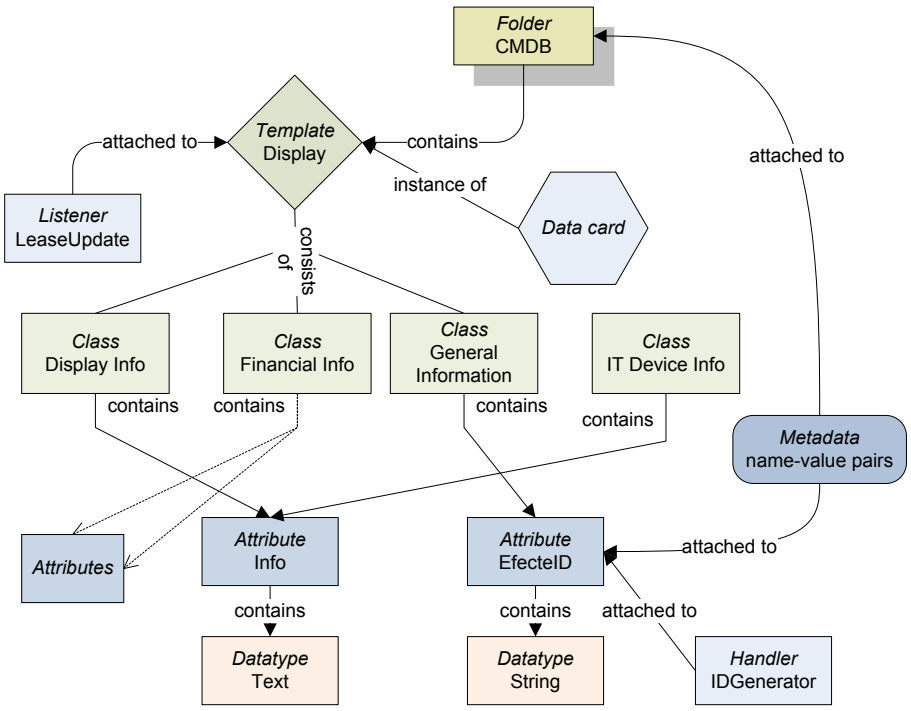


Figure 12: Efecte system contains *data cards* that are instances of templates that themselves consist of groups (consisting of attributes).

5.2 Efecte Web Application

Efecte has been in development since 1999 and during this time the landscape of web technologies has changed drastically. The most significant changes have been the introduction of Ajax (Section 4.2.3) and renewed interest by browser developers on adhering to the web standards. The first one allowed easy sending and receiving data from the web server without reloading pages and thus allowed more dynamic web applications and the second one reduced web developers' work required for supporting different browsers.

Being developed incrementally without any complete rewrite, Efecte is an example of this history: new technologies were adopted after they had been proven suitable elsewhere and new features were developed using those new technologies, however old features rarely were rewritten using the newer technology.

The user interface of Efecte consists of three types of web pages: static pages, dynamic pages created by JSP and dynamic pages created by IT Mill Toolkit, a third party toolkit for creating Ajax interfaces. Only the dynamic pages are interesting from security point of view.

Mapping of different requests to Efecte is done using servlet mapping (Figure 13), where patterns such as `/efecte/*` maps all requests that match that pattern to a specific servlet. Servlets in Efecte's case then map to different front ends: one

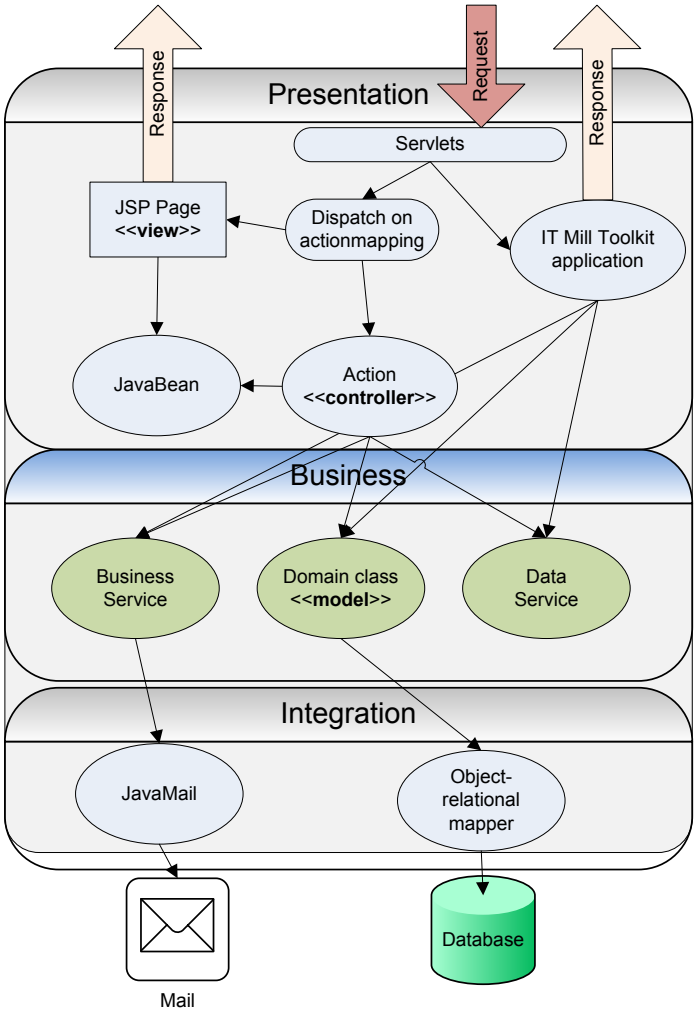


Figure 13: Processing of one request in Efecte.

for normal Efecte, one for web-based integration and a few for different IT Mill toolkit-based applications.

5.2.1 JavaServer Pages in Efecte

JavaServer pages is a Java technology developed by Sun Microsystems. It is a template engine that allows intermixing of HTML with JSP *tags* in a "JSP file" that it then turns into HTML. Adding a tag into a JSP file causes actions defined in the Java code that the tag is an abstraction of. These actions usually consist of creation of HTML based on parameters given to the tag or on the HTTP request or both. In principle, all development of JSP files should use only HTML and these tags with developers creating new tags to support new features.

In practice, many JSP files also contain Java code within special JSP tags that allow writing of inline Java code. This inline Java is written when creating a new JSP

tag is seen to cumbersome for the problem in hand but often the same code gets written again and again in some other JSP files without creation of a JSP tag to replace them all.

After normal Efecte gets an request, a controller is invoked to select what JSP or Java class should then be called to create a response. Controller selects the JSP or Java class based on a XML configuration file and if a suitable mapping is found based on the request, invokes it ("Dispatch on actionmapping" in Figure 13), otherwise generating an error page.

The invoked Java classes and JSP files are called *actions*. The actions present some semantic action such as "send mail to this address" and to carry out the action, call other code in Efecte, retrieve and modify data, etc (Business layer in Figure 13). All parameters of the HTTP request are available for the action in an object called "request" that can also be used to save data to. After dispatch to action is complete, controller looks up if there is a result page to show. This result page then gives user feedback of her request ("mail was send successfully") and is dynamically created from a JSP using data the action earlier has saved in the request.

All actions are not supposed to be used by all users. Actions such as displaying Efecte log files or showing connected users are available only to Efecte administrators, a condition checked at the start of an Action.

5.2.2 IT Mill toolkit

IT Mill toolkit is an open-source web application framework based on the popular Google Widget Toolkit (GWT). IT Mill toolkit, henceforth only "the toolkit", is a framework that allows developers to build dynamic web applications using only Java, with the toolkit taking care of dynamic creation of JavaScript and HTML. Toolkit offers the same approach to graphical user interface (GUI) creation as desktop GUI toolkits: it is event-driven with centralized event-queue and offers ready-made widgets for composing displayed components such as windows and buttons.

Two main benefits offered by the toolkit are these ready-made widgets and the toolkit's security features. The security benefits the toolkit offers are two-fold; the toolkit validates all actions on the server, preventing users from tampering with the data and toolkit offers a centralized place for protections against attacks such as XSS and CSRF.

Summary

Efecte uses two different technologies for generating its user interface. Although IT Mill toolkit is poised to replace JSP-based part of Efecte, the work on this area has not progressed very fast. This means that for the time being, all security issues need to be considered in these two scenarios.

5.2.3 Deployment

In addition to the Efecte web application part, an installed Efecte system consists of a database and usually of a device information source. Currently the only supported database is Microsoft's SQL Server. For device information, Efecte supports out-of-the-box Discovery, a tool by Centennial software to track computers and other network devices, or Inspector, an in-house software for the same purpose.

All Efecte instances are deployed in the customer's intranet and connected to various services offered there, (e.g. directory service for users) as shown in Figure 14. It is

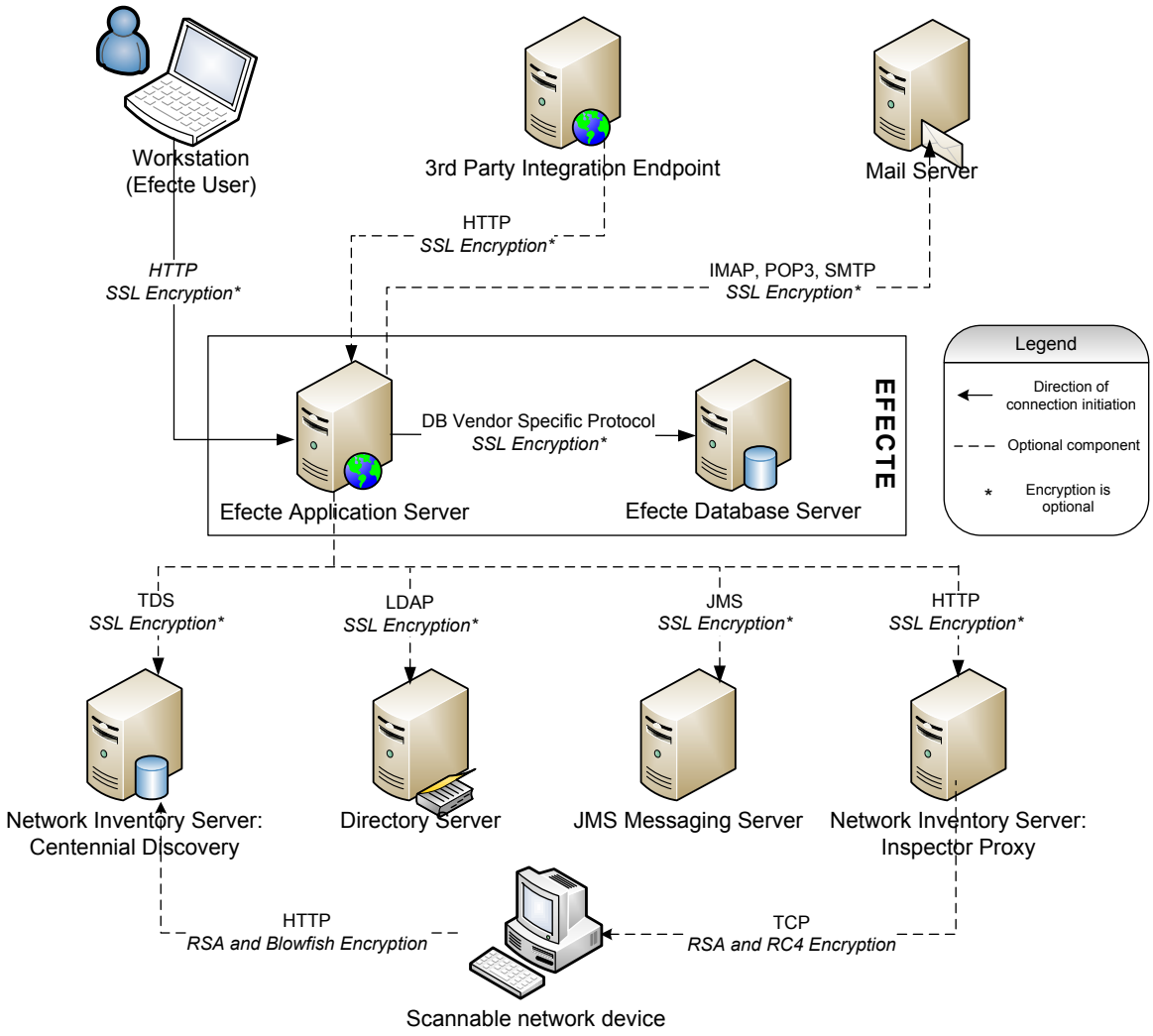


Figure 14: Deployment of an Efecte system.

noteworthy that no traffic of Efecte system, save the traffic between workstations and network inventory servers (Centennial or Inspector), is encrypted by default as intranet is assumed to be trustworthy.

Assumption 1 *No one untrustworthy can read or alter messages in the network Efecte is installed in.*

Deployment as such consists of installing Efecte, installing device inventory server and of configuration of Efecte and possible integrations to other systems. In the first step, Efecte installer copies an application server called Resin, a Java runtime and Efecte the web application to the server that is used to run Efecte. The installer also sets up the database used to store Efecte data and configures Resin the application server to run as a Windows service so it is automatically restarted in a case of a crash.

Installing the inventory server is very straightforward and the nature of the service does not leave room for many vulnerabilities. The only observed oddity was that both Efecte and Inspector agents are installed to run with too large privileges compared to privileges required, discussed further in Section 6.6.

The last step is the configuration of Efecte and its integration with other services. Configuration of Efecte means both mapping users IT domain to Efecte and setting permissions. Integration with other services means selecting services that are used to import data to Efecte or that Efecte exports data to and defining required conversions. Greatest concerns here are wrong or misbehaving services and ad-hoc Jython scripts written by consultants doing the deployment.

The data the other services import to Efecte is trusted and thus can cause integrity problems as the data only verified to have a proper structure. Jython scripts are used to implement a customer specific needs not realizable through normal means (handlers and listeners) but their power (scripts are trusted code) combined with lack of security awareness can lead to vulnerabilities.

Efecte the company tries to assure correct installation and configuration of every Efecte system deployment by training the consultants who perform the installation and configuration: whenever the the installation process is changed, training sessions are held and support material is updated.

5.3 Efecte access control

5.3.1 Authentication

Efecte stores state of the application in the server. When user's browser connects to Efecte, Efecte first sends it a cookie named `JSESSIONID` identifying the session Efecte has created, and forwards the browser to the login page for authentication.

Efecte has been deployed in many different environments with different authentication requirements. This has led to development of authentication framework supporting different authentication methods through different classes implementing `PluggableAuthenticator` interface as shown in Figure 15. Most commonly, user is authenticated by a typed password or by NTLM authentication in case of Windows users. Authenticated user is then bound to session and later identified from a session cookie the user's browser sends. Efecte's security is based on the assumption that only the user (or her browser) and server can know the value of the session cookie.

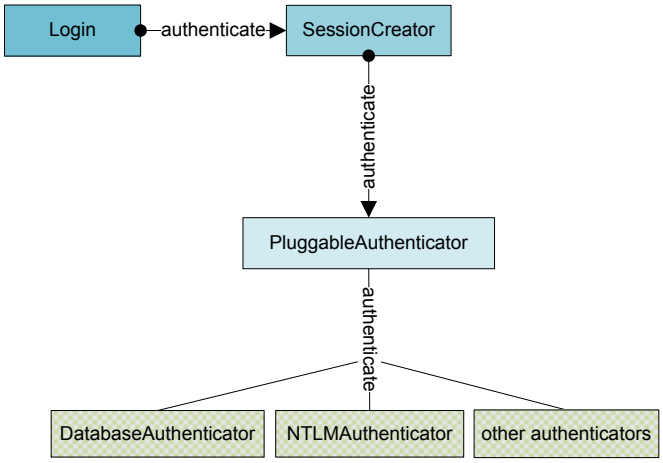


Figure 15: Efecte's authentication delegation from Login to actual authenticator.

Assumption 2 *User's* `JSESSIONID` *is only known to the user and Efecte.*

Such assumption is justified as long as traffic in the network cannot be snooped (Assumption 1) and user's cookie cannot be divulged through some other means such as through XSS vulnerabilities (Section 4.3.5) or by predicting its value.

5.3.2 Authorization

In brief, Efecte access control model is quite near that of role-based access control model (discussed in Section 3.2.5), augmented with trusted procedures such as the ones found in Clark-Wilson model and Biba's model with ring property (Sections 3.2.4 and 3.2.3 respectively).

Permissions in Efecte are divided in three different groups: product permissions, administrative permissions and data card permissions. All authorization in Efecte is done for $(user, action)$ tuple, with each user belonging to zero or more roles and action depending on permission group. In this role-based access control model, each role has a tuple of permissions containing all product, administrative and data card permissions:

$$P = (P_{datacards}, P_{products}, P_{admin}), \text{ where}$$

$$P_{datacards} = (P_{templates}, P_{folders}, P_{attributes})$$

Table 5 lists what types of permissions each permission group contains. Effective permissions in each of the permission groups are union of permissions of each of the roles the user has.

The most straightforward of the permission groups are data card permissions that work quite like file and directory permissions in operating systems.

Table 5: Different permission groups in Efecte

Permission group	possible modes	targets
Product permissions	read $\oplus$ write $\oplus$ none	different products Efecte Web API
Administrative permissions	$true \oplus false$	edit users edit static values edit templates edit folders import data cards
Data card permissions	create | read | update | delete	templates folders attributes

Data card permissions group contains four different permissions types of access permission - "read", "write", "update" and "delete" - that correspond to so-called "CRUD"-operations [Kil90] of database-level. Efecte does not allow blind writes, updates or deletes so having permission to any of these operations also gives permission to read the data card. Access control is resolved on sub data card granularity: access to a data card is based on $P_{templates}$ but access to different attributes that the data cards consists of is based on $P_{attributes}$. So user in one role may be able to see and edit parts of a data card that are not shown to some users in another role.

Still, this level of granularity is not enough in situations where each user would need some individual permissions. For example, each help desk employee has her own set of cases assigned to her and she needs to be able to write on those so that she can close the cases and her progress can be tracked. Yet we cannot give her write access to all help desk cases as that would create integrity problems. The solution is to grant discretionary access to the user.

This problem could be solved in two ways: either within the role-based access control model, adding each user a unique personal "role" with permissions specific only to this user or outside it by adding another mechanism to gain permissions in addition to roles. In Efecte, the latter route was taken and the finer-grained permissions are implemented using *Elevated User Permissions* or EUPs for short that allow giving users more permissions to a certain card, such as these help desk cards. EUPs included, the effective permissions a user has on arbitrary data card D are given by $P(D) = P_{elevated}(D) \cup P_{role_1}(D) \cdots \cup P_{role_n}(D)$.

Product permissions either deny all permissions or give permission to view ("read") or fully use ("write") some parts of Efecte system. Product permissions limit data card permissions as a user can only access data cards that are created from templates residing in products the user can access.

Administrative permissions is the other interesting group of permissions. All administrative permissions are constrained by data card permissions and product permissions: if a user cannot access a data card or folder, the user cannot edit it

either.

Edit users allows editing of users *if* editing of the user data card is allowed otherwise: a user with "edit users"-permission can edit other users only if the user can edit the folder the user data cards reside in.

Edit templates grants permission to create, update and delete templates and attributes' and templates' permissions in products user has access to.

Edit static values allows a user to edit all static values. Access to static values is not constrained by other permissions.

Edit folders grants permission to edit folders user can read beforehand, allowing the user set what templates are allowed in a folder and what access rights roles have to folders.

Import data cards grants permission to import data cards to Efecte, to update old cards and to create new cards.

In addition to these access controls, each user is designated on one of three **user levels**: read-only, normal or *root*. When user level is set to normal user, all access controls work as described before. With read-only level, all permissions to modify (create, update, delete) are revoked and with root level, all access control checks are by-passed.

Like most settings in Efecte, users and roles are first created during the deployment phase. Customers can then later add new new users and roles by themselves but this is not done constantly so it is safe to assume that permissions are configured in line with company's security policy.

Assumption 3 *All roles and users have properly configured access rights.*

5.4 Approach

As a 2001 workshop on information assurance found that no single metric can give an objective picture of the trustworthiness of a system [VHS03], we will use multiple metrics to evaluate Efecte's security. Our approach to evaluating Efecte's security consists of following steps:

1. List and analyze threats and risks affecting Efecte.
2. Get familiar with Efecte's architecture and do architectural risk analysis, as in Section 3.4.2.
3. Analyze Efecte's data flow in the spirit of attack surface measurement, as in Section 3.4.3, but without comparisons.

4. Do white-box code review and penetration testing on the code directly exposed to attackers.

5. Repeat phases 2 - 4 to most important third party components, as thoroughly at possible.

It needs to be noted that problem here is the same as in all software testing, namely that these methods can only be used to find defects but not to show lack of them; to give assurance but not proof of security.

5.5 Threats and risks

First thing done was an enumeration and analysis on threats against Efecte and their effect on different customers. As noted earlier, Efecte is predominantly used to model IT domain in different companies, but because of the dynamic object model used use of Efecte is not limited only to that domain. Analyzing threats common to all customers is impossible without knowing each customer's current use cases and deployment situation, which we lack. Therefore this analysis is necessarily based on our recommended deployment pattern described in 5.2 and on the most common use cases.

Based on this, we define our exemplar customer as follows: IT-department of a company, running Efecte on Windows platform with a few dozen simultaneous users. Such a customer uses Efecte to maintain its IT infrastructure and possibly other things such as contract stock. From these assumptions we can derive following assumptions on Efecte and its deployment.

1. Efecte is installed in customer's intranet and is reachable only within this network.

2. Efecte contains data that is valuable but not critical to customer. Loss of confidentiality is not as bad as loss of integrity and availability.

3. In case of outsider attack, Efecte will be the first target only if it will make other attacks easier.

4. Customers use Efecte in a supported environment.

First assumption means that Efecte is partly protected through physical security and in case this physical security is breached, second assumption says that the customer will suffer greater loss from compromise of other assets. Third assumption is tied to the second: Efecte will not be primary target of attacks as it will not contain as valuable information as customer's other systems but it can still be first target if attacking Efecte will help with attacking primary targets.

Earlier, Efecte was installed in dozens of different environments but since 2008 there has been a policy in effect concerning environments Efecte should be installed to

and what browsers are supported. Currently, supported installation environments are Windows Server 2003 and 2008 and of browsers, Internet Explorer 7 and Mozilla Firefox 3 are supported. Supporting only a few browsers makes security evaluation easier given how much browsers differ on small details (for a throughout study, see [Zal09]) and does not restrict possible users significantly as those two browsers currently have most of the market share.

Having Efecte only in intranet leaves two possible groups of attackers: insiders and determined outsiders leaving out casual outsiders that target sites in the Internet. This combined with assumptions on Efecte usage mean that attacks that make it possible for outsiders to reach Efecte and attacks that cause loss of integrity and confidentiality are the most severe types. Even though availability was valued in the assumptions about Efecte usage, availability attacks from the insiders are easier to detect and stop than attacks on confidentiality and integrity.

Based on these assumptions, we first generated and analyzed threats based on possible attacks with attack trees. These attacks were the substantiated with architectural risk analysis that brought up our dependency on some core third party components (discussed further in Sections 6.2 and 6.4).

5.5.1 Attack trees

To generate and concretize threats, attack trees were employed. However, results of the attack tree analysis are not published here. Publishing the results could give possible attackers information on what angles of attack have possibly been left uncovered and reap virtually no benefits. Instead, problems encountered in creating attack trees are described.

Creation of attack trees was not easy. Largest obstacle in generating attack trees was that the lack of control on deployment environment. Even though an ideal environment for security analysis was defined, it was not controlled enough to allow precise estimation of what attacks are possible and how possible they are. Especially hard was coming up with a realistic goal as Efecte's usage and data stored there varies. Only goal we used was "get access to data in Efecte", whereas actual attackers goals will vary depending on the organizations attacked.

A better way to employ attack trees would be to do a security analysis at each deployment environment and see what would attacker's goals be there. If attack tree analysis would then show that viewing information only available in Efecte were attacker's goal, customer and Efecte could make defensive preparations in Efecte deployment and the environment it is deployed in.

Nevertheless, attack tree analysis had some benefits: it pinpointed areas in Efecte where Efecte is most vulnerable and made explicit the assumptions on which Efecte's security is built.

5.5.2 Architectural risk analysis

Architectural risk analysis (Section 3.4.2) starts with a creation of high-level view of the system. Since there was no single document giving a one-page view of the system, creation of the view was considered. As this analysis was done with the chief architect of Efecte, such a documentation was ruled as unnecessary, based on the fact that all information was readily available through questioning and documents describing architecture partially.

With hindsight, this was wrong decision. Although the information was freely available, its processing was harder than what would've been with a good one-page picture. This may be the most important cause why some of the process' parts did not seem to work out.

First part of the process, **attack resistance analysis**, yielded a few results. As a result of missing the one-page picture, some parts of the system might have been overlooked. As there was little information of historical risks, most of the process was abuse case analysis: enumerating different parts of the system and thinking how they could be abused. As my experience on abusing systems was quite limited, usage of weakness and vulnerability listings such as OWASP top 10 Java EE vulnerabilities and especially CWE list of most dangerous programming errors were a great help.

Second part of the process, **ambiguity analysis**, was still harder. McGraw states: "This process, by definition, requires at least two analysts (the more the merrier) and some amount of experience". As noted earlier, the whole architectural risk analysis — as probably all risk analysis — requires experienced analysts and there were none available.

Ambiguity analysis was done in a quite ad-hoc manner with reading parts of code and documentation to get an idea how the system supposedly works and then questioning chief architect on how this part was supposed to work. The analysis brought up parts of the system that seemed error-prone but did not uncover any new weaknesses. Proneness to error was almost always result of needlessly complex implementation of control mechanisms.

Last part of the process, **weakness analysis** was the most painless part. Weakness analysis tries to assess what trusted parts of the system could contain weaknesses and how this misplaced trust would affect the whole system. Efecte rests on quite a large stack of software:

1. On third party libraries and frameworks. Most of the libraries are used only for a small part but then a few pervasive frameworks touch many levels of Efecte.

2. On Java application server. Currently supported application server is Resin from Caucho Technology inc. but some customers run their own application servers.

3. On SQL database server. As seen in Figure 13, all integrity of Efecte's data is

based on database's integrity. Efecte currently supports only Microsoft SQL Server.

4. On Java virtual machine (JVM). Although some customers run Efecte on other JVMs, our current supported JVM is Sun's release 6.

5. On the operating system running all preceding software. Operating system is some version of Windows, most often Windows 2003 server.

Of this stack of software trusted, it is customers task to keep their operating system and database server secure. They are "blindly" trusted, meaning that we will not take their impact on security into account as we cannot do much on them. This left application server, Java virtual machine and third party libraries still to be considered.

Any security problems in JVM and application server would clearly cause problems also in Efecte if the problem was not strictly bound to local access only. With libraries the results were not as clear for usage and pervasiveness of the libraries varied greatly. Guidelines adopted for estimating libraries' importance were to check in how many files this library was used and to check was the library used directly ("call a function and get results") or indirectly ("set up environment for the library to work"). Frameworks work in this indirect way and touch many more parts than a library for drawing charts and are thus "trusted" more than normal libraries.

This analysis both clarified relative importance of different third party components and made it clear that some tool to track problems in third party components was needed. Although security companies such as Secunia offer services for tracking third party vulnerabilities, we decided to implement a simple tracking tool just get started with following these vulnerabilities. This tool is described in Section 6.2.

5.6 Vulnerability analysis

To find vulnerabilities and places that might be vulnerable, Efecte's attack surface was first measured and then code on that surface was reviewed and penetration tested. These phases were then repeated in smaller scale for two important third party components of Efecte: Resin application server and IT Mill's GUI toolkit.

5.6.1 Attack surface analysis

Attack surface measurement, described in Section 3.4.3, is a metric meant for comparison of two or more pieces of software. As we are not interested in comparing Efecte's security to some competitor's product, we adapt only the general idea of assessing security through system's simplicity and through the surface exposed to attackers.

The first few articles on attack surface measurement by Howard et al. [HPW03] and by Manadhata et al. [MTMW07] concentrated on measuring attack surface of

different program written in C which made them a bit less useful given that Efecte is written mostly in Java. Manadhata et al. had, however, also later applied their methods to Java-based software at SAP AG [MKW08]. Based on these articles, measuring attack surface of a system was easy in principle:

1. Generate call graph for the whole system.
2. Identify entry and exit points based on the call graph and I/O-operations used in calls.
3. Assign damage potential-effort ratios.

First problem encountered was that the tool used by Manadhata et al. to generate call graphs, TACLE, did not support newer versions of Java. This led to research for other call graph generators but none of the tested were suitable for use. Hence the idea to use call graphs was discarded and instead potential entry and exit points were found by hand. Manual attack surface analysis has the downsides of not being easily repeatable and possibly missing some parts of the surface. Consequentially, state of call graph generating tools should be checked frequently to see if any tool would allow easy creation of attack surface measuring tool.

Inspection of code and architectural documents (such as deployment diagram in Figure 14) revealed that Efecte receives data from four main sources: from HTML-forms, from web services, from data imports and from its database and mail. Database is seen as a part of the system and thus considered trusted. Import being an operation requiring special administrative privileges (Section 5.3.2), this left two sources that can be used by most of the users and one source that can only be used by the selected few allowed to do imports.

Information flows out of Efecte to three destinations: to web pages shown to users, to database and to other systems in case of integration interfaces such as XML export and JMS messages. As Efecte's state ultimately resides in the database, all writes to it need special attention.

Based on these observations, we started to trawl through all potential entry and exit points. Points were quickly classified to three informal classes, "trivial"; "normal" and "fishy", with "normal" being code that did not seem too complex and "fishy" was tangled code that looked hard to understand and therefore, perhaps, hard to get right. "Fishy"-class also contained points where standard abstractions Efecte provides for handling database or HTML generation were not used signifying special circumstances and special chance to make mistakes. If code seemed to handle files, an extra note was added to remind of checking the code better later as handling of files is a common source of vulnerabilities.

Classification and checking if entry- or exit-point wrote data somewhere else than to the immediate response constituted the damage potential rating. There were three effort levels, based on how easy the functionality was to find without source code: "easy" meaning all users can see it, "medium" meaning that only Efecte

administrators can see the functionality and "hard" meaning that even though the functionality was available, it was not used anywhere. Entries in last category are discussed more in Section 6.1.

This list of entry and exit points together with assigned damage potential-effort ratios shows where vulnerabilities would have the greatest impact and can be used to prioritize both reviews and mitigation efforts. The review part is taken to extreme in the next step, penetration testing, where risky parts of the system are exposed to attacks.

5.6.2 White-box Penetration testing

Whereas functional testing is "testing for positives", testing that some functionality exists and works properly, penetration testing [Bis07] is "testing for negatives". Problem with testing for negatives is that there are virtually infinite ways for system to fail and only limited ways to succeed. All security testing tries to create assurance so that we can be assured that system's security is *precise*, allowing only all secure states and not *broad*.

So, question of whether penetration testing should be done at all is a valid question. McGraw lists common critique on penetration testing [McG06]:

- Penetration testing is done just to allow managers say security issues have been handled.
- Penetration testing represents a too late approach to security as when done late, fundamental problems found are very expensive to fix at that stage. This leads to quick band-aiding over symptoms instead of removing the cause.
- Penetration testing is usually done by an overzealous information security team not connected to normal developers, resulting both sides getting angry as developers think security team reports insignificant problems and security team thinks developers do not take their reports seriously.

All these points may be valid in some environments but in the case of Efecte, only the second point might be true. One important tenet of penetration testing, that our testing slightly violated, is that it happens in the deployment environment so problems rising from the environment and configuration can be found. The violation in our approach was to test on developer's machine instead of setting up a separate computer to penetrate.

Results of this violation are threefold: finding of configuration problems is limited and network-dependent problems cannot be found at all, but finding other problems is faster as program execution can be followed better with debugger and other developer tools. As Efecte is configured individually for every customer, we can only find problem rising from the basic configuration that is used as the basis of customer's configuration. This is the same configuration that we can test; what we cannot test is interplay of basic configuration and customized configurations.

Inability to find network dependent problems is not a great loss as such problems were known up front (see Section 6.3). Upside of allowing trace execution paths with debugger more than made up for the downsides. If a penetration attempt "almost succeeded" meaning that it was only blocked by a control at some final stage, debugger allowed us to see if this control would block other attempts too or would the attack succeed, speeding up the testing.

After penetration testing for the list from attack surface analysis was done, all uncovered vulnerabilities were given a severity estimate. The estimating method was inspired by common vulnerability scoring system [MSR+07] that is used to score vulnerabilities in mass-marketed software. We discarded the temporal and environmental dimensions from the scoring and only focused on the base metrics: what does the vulnerability mean for Efecte. Our scoring included five factors: impact on confidentiality, on integrity and on availability, required level of access to Efecte and an estimate of how easy the vulnerability is to find.

As it is not wise to rely on safety of the source for security, the last factor was used only as a tie-breaker. White-box based penetration testing brought two benefits: a list of prioritized weaknesses and assurance that other entry- and exit-points are quite safe. This prioritized list can be further mined for similarities that can lead to overlooked problems in architecture or problems in control implementations.

5.6.3 Significant third parties

There are two third party components in Efecte that constitute a significant amount of Efecte's attack surface: application server Resin and GUI toolkit IT Mill toolkit.

Resin

Resin is a Java application server from Caucho Technology used to run Efecte. Resin comes in two versions, "Resin Open Source" released under GPLv2 license and commercial "Resin professional" with latter version being used in Efecte installations. With even the "Resin Open Source" consisting of 650,000 lines of Java source code and of tens of thousands source code lines in other languages, nothing but a cursory glance at the most probable sources of security problems was possible.

One such a source is generation of session identifiers. Tomcat, a competing application server, has had multiple problems in its handling of session identifiers, documented in various vulnerability advisories, and a session identifier generation attack by Gutterman and Malkhi [GM05].

In their attack, Gutterman and Malkhi were able to decipher seed value used to generate session identifiers one day of calculation. With seed value, all session identifiers generated by Tomcat are known and new sessions created by other users can be hijacked at leisure.

The attack works because instead of 160 bits SHA-1-based pseudorandom number generator (PRNG) being unknown, only exact startup time of the server is unknown. If the server were started within a day, this would give only 2^{26} ms to find the

correct seed from and were the server started within a year, 2^{35} milliseconds. Now what attacker has to do is initialize the random number generator for each of the milliseconds and generate random number until a session identifier that attacker has retrieved from the server is returned (seed is found) or estimated maximum amount of session identifiers the server could have generated is surpassed (incorrect seed).

Thus, there was a clear need to see if the session identifiers used by Resin could be compromised in the same way. Source code inspection was done using "Resin Open Source" source code which revealed that Resin uses a same method to generate session identifiers as Tomcat except that Resin appends current time string to the generated random number string to create the session identifier. Based on packages included within "Resin professional", it uses the same method for generating session identifiers.

This all points to the direction that Resin is not as vulnerable as Tomcat. How hard an attack on Resin is, remains to be seen as application of Gutterman and Malkhi attack was postponed as there were more urgent problems found in Efecte.

IT Mill toolkit

IT Mill toolkit is framework for creating dynamic web applications using only Java. As such it in as ideal tool for Efecte as it allowing developers to use their knowledge of Java for development and outsource problems of multi-browser compatibilities to the toolkit developers.

Two main threats were that the toolkit would have XSS or CSRF vulnerabilities. Much of IT Mill user interface components are based on Google Web Toolkit (GWT) and based on its popularity and Google's testing practices, GWT was estimated to be an unlikely source for vulnerabilities. Thus, vulnerabilities were only searched in IT Mill's custom components. To find vulnerabilities, source code of these custom components was then scanned for code that either creates HTML dynamically or uses JavaScript. No dubious components were found and as GWT has been used extensively for three years, we chose to trust its XSS protection mechanisms.

Cross-site request forgeries are a different story: GWT does not directly protect against CSRF. However, the toolkit includes its own CSRF protection where every session has an additional, cryptographically secure, random number connected to it. This random number is then submitted with each toolkit request to the server that checks if it matches the value stored in session.

These protections means that Efecte Corporation can trust IT Mill toolkit not to be a source of either of these vulnerabilities.

6 Flaws and Solutions

Based on the approach taken in Section 5, this section will report some of the problems found in Efecte security. Each problem will be first described, then if needed, evaluated and lastly, mitigative actions and their possible implementation are discussed.

6.1 Old testing code in production application

There were multiple cases where code that developers had written just for testing purposes was found in production. As these pieces of code were not written security-consciously, quite contrary as the pieces of code were supposed to help testing, these pieces all contained vulnerabilities of varying severity. The pieces, having laid dormant for years, were mostly found during the attack surface analysis and white-box testing.

During the early years, Efecte did not have any clear process on how new features were developed and shipped. Instead, one developer might have developed a feature requested by a customer and if the developer was satisfied with the code, the code was checked into version control and shipped to customers. This led to situations where development code never intended to be included in a shipped application nevertheless was included and to situations where the developer did not consider how the developed feature would have affected Efecte's security.

6.1.1 Mitigation: better process

Development methodology in Efecte has changed significantly since the early days. Nowadays, there is a clear process for how the development is done. In this process the whole development team "owns" the code and code review is done for all new features and changes. This ownership of code means that there are fewer places in code that are developed by just one developer, leading to fewer defects and vulnerabilities because more eyes see every part of the code. All development happens in separate branches of version control system and before this code is combined with the main code line, there is a code review. This code review is done by a developer who has not taken part in development of this specific feature or change so the reviewer brings a pair of fresh eyes to the process and catches problems that have evaded the developing team.

This process has virtually eliminated the possibility of including features not meant for production in the shipped application. Furthermore, regular code reviews try to assure that new features with questionable security do not get into code base.

6.2 Vulnerabilities in third party components

As Efecte uses over forty different third party libraries, being aware of latests problems in code from those sources was cumbersome and was not done systematically. This meant that when developers of these third party libraries announced their fixes to vulnerabilities (disclosing the problem at the same time), it would often take from days to weeks before this was noticed by Efecte developers. This would give possible attackers time to use these disclosed vulnerabilities before Efecte code base and customers using Efecte were upgraded to the fixed library version.

Although there have been no records of such attacks, during architectural risk analysis (Sections 3.4.2 and 5.5.2) better tracking of third party component vulnerabilities was deemed very important.

6.2.1 Mitigation: tracking third party bug reports

First the list of used third party libraries was brought up-to-date. Next we acquainted ourselves with how development teams behind these libraries report their bugs, vulnerabilities and solutions. Most of the libraries used are open-source and happily many open-source teams use either Mozilla Bugzilla, Atlassian JIRA or Sourceforge.net's own tool as their bug tracking tool. All of these tools support filtering of bugs and fixes by severity and additionally, Bugzilla and JIRA support filtering by version which allowed creation of views containing only bugs in version that interest us and that were severe enough. These views were then exported as

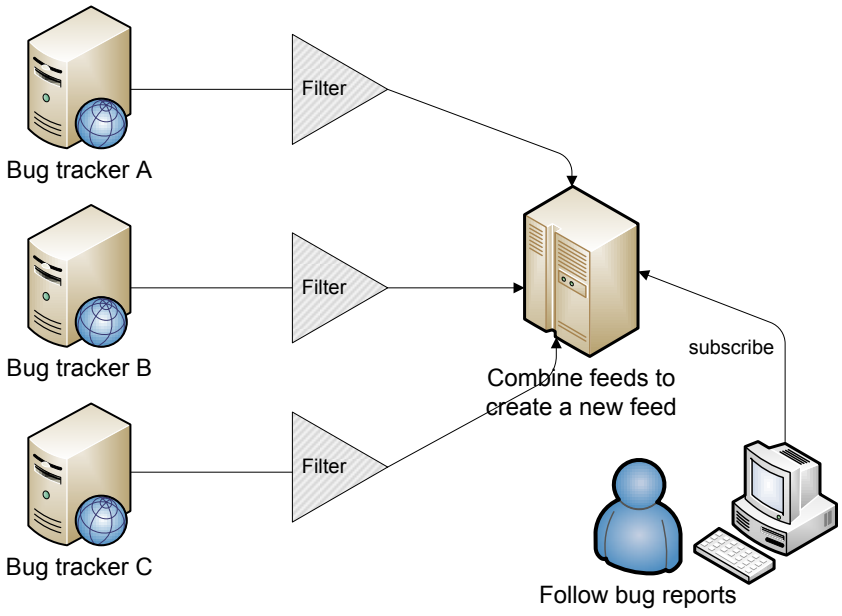


Figure 16: Simple third party vulnerability tracking

web feeds and the feeds were combined using a custom tool producing the final feed that could be subscribed to (see Figure 16).

For libraries that only publish changelogs or corresponding text items where changes are announced, a partial solution is to implement custom parsing logic. Unfortunately, not all libraries publish the security vulnerabilities they have fixed making decision of when to move to a new version hard. Their reasoning is that users of the old versions are thus protected from exploitation.

6.3 Unencrypted traffic

By default, Efecte and Inspector allow unencrypted or barely encrypted traffic. This delegates final responsibility for all Efecte's security to impenetrability of the network in which Efecte is deployed. Even though companies often keep routing equipment and switches in safe places, network still often succumb to other attack such as ARP poisoning [Wha01, Zal05]. ARP poisoning allows an attacker to pretend that she is Efecte, Efecte's user or both, making session hijacking a triviality or even worse, intercept traffic between Efecte and the database server.

A lesser problem is the very weak encryption used in Inspect. Inspector proxy, the server gathering data from different computers, uses 512 bit RSA public key for the handshake used to create 40 bit RC4 session key. RSA public key with length of 512 bits requires 8400 MIPS years [CLR+00] to factor and currently top-notch desktop processor deliver 76000 MIPS meaning that one such processor would factor 512 bit RSA key in approximately forty days. Special hardware or multiple computers could bring this cheaply within a day's range.

Even more compromisingly, Biham and Carmeli have shown that 40 bit RC4 session key can be broken in 0.02 seconds [BC08] using a single desktop computer. This means that encryption used by Inspector is only a slight hindrance, not a protection by any means even against an attacker without any special resources.

Thus, it is safe to assume that all traffic between Inspector and workstations is listenable. Therefore either encryption used must be enhanced or it must be assumed that no confidential information is passed between Inspector proxy and the scanned device. The only information that can reasonable be expected to be confidential is list of installed software (what the computer's user and company do) and the list of users that have logged into the system. Thus Efecte the company chose the latter route, considering Inspector traffic non-confidential.

6.3.1 Evaluation

Traffic that is not strongly encrypted is susceptible to insider attacks and to outsiders that have gained an entry to company's network. Traffic in such network is snoopable given one of the following weak assumptions: used network equipment does not prevent ARP poisoning, static IP addresses are used instead of DHCP which disables network equipments' ARP poisoning prevention or physical snooping of the traffic is possible either by using cables or wireless LAN.

There are virtually no reasons for not using encryption. Given the possible attacks the unencrypted traffic allows, combined with ease of turning encryption on, there is only one choice of action.

6.3.2 Solution: enabling encryption

For the first problem, the solution is simple and straightforward. Efecte needs to be configured to only accept HTTPS connections, instantly upgrading connections from HTTP to HTTPS before allowing other traffic. To prevent spoofing with self-signed certificate[4], no self-signed certificates should be used with Efecte. In cases when a self-signed certificate is used, such certificate should be delivered to users beforehand so their browsers will warn if the certificate changes. Additionally, as database holds the confidential material and admin password to Efecte, SSL needs to be enabled on Efecte's database driver.

Additional care should be taken to confirm that Efecte's session identifiers are only send with encrypted traffic to prevent attacks where browsers are tricked to make unencrypted requests to Efecte.

As for the Inspector problem of very weak encryption, this is not usually a problem. As Inspector just sends information about the installed hardware and software, snooping such traffic is rarely a problem. However, if customers feel that their traffic needs to be encrypted with a credible encryption, they are offered another inventory tool called Discovery. Discovery supports 1024 bit RSA keys combined with 128 bit Blowfish symmetric cipher, thus offering a significantly better level of security.

6.4 Resin session id generation

Resin session identifier generation seems vulnerable to the same attack as Tomcat's session identifier generation (see Section 5.6.3). However, two mitigating factors are present: Resin adds the time it receives a request to the generated random number and reuses session identifiers.

Adding the time of the request to the generated random number adds more information the attacker need to infer so the attacker needs to make regular connections to Efecte to estimate the time when a new session identifier has been created for a user.

If the user of Efecte still has a cookie in the browser (the browser has not been shut down), the user's browser sends it to Efecte when connecting to Efecte. Even if the user's session at Efecte has been closed, Resin reuses this cookie and the session identifier it contains. This reuse of session identifier lessens the opportunities

[4]A certificate (Section 4.1.2) that is not signed by a commonly trusted certification authority. Everyone can self-sign certificates and thus deliver public keys but this gives no assurance that they key is from who it is purported to be from.

available to the attacker and thus requires a longer period of active attack.

6.4.1 Evaluation

Attack seems feasible but would generate a noticeable amount of connections that offer a slightly changed session identifiers. Were the attacker to check is Efecte had issued a new session identifier every five seconds, the attacker still would need to make at most $5\ s * 1\ connection/ms = 5000$ connections to the server in order to find the correct time stamp. This would be noticeable, had Efecte a ' detection system to detect the attack.

Feasibility of the attack seems especially good when Efecte has just been started as both the uncertainty about seed value and the amount of session identifiers generated is low. Additionally, as the attack is not based on anything new, there is no reason to believe that a determined attacker would not be able to come up with this attack and test its functioning.

6.4.2 Mitigation ideas

The whole attack stems from the lack of real entropy source for pseudorandom number generator. SHA-1 based PRNG has 160 bits of internal state but only some part of it (2^{20} to 2^{36}) is unknown to the attacker. Were Resin able to make the inner state less predictable, for example by using operating system's entropy pool, the attack would be completely foiled. Even doubling the entropy contained (to 2^{72}) would make the attack infeasible. This is, however, something that Caucho Technology, the developers of Resin, have to do. This possible vulnerability has been reported to them.

A fix that could be implemented in Efecte would be to save sufficiently (say 10000) large amount of last session identifiers offered and the hosts offering them. In case a host offers enough non-existent session identifiers, an alarm could be issued to the Efecte administrators. This fix assumes that attack would come from only a few computers, a valid assumption in a corporate intranet.

6.5 Path traversal

Path traversal vulnerability is a subclass of vulnerabilities where a developer exposes a reference to some object internal to the web application. In path traversal that object is a some file or directory, as in the following example:

1. Path to a file is read from request.
2. A handle to the file is constructed based on this path.
3. An action is taken on the file.

Often, there is a step between first and second called “improperly check input” where the file is supposedly checked to be within some specific directory as in Listing 10.

Listing 10: Path traversal vulnerability

```
String path = request.getParameter("path_to_file");
if (path.startsWith("good_directory")) {
  File file = new File(path);
  //Do something with the file
} else {
  //Take action against a possible attack
}
```

While this approach of checking works against naive attacks such as setting request parameters ‘`path_to_file` to `c:\\windows\\secrets.txt` it can be bypassed by relative paths such as `good_directory\\..\\..\\windows\\secrets.txt`.

6.5.1 Discovery

During attack surface analysis phase (Section 5.6.1), source code files that contained entry and exit points handling both files and request data were marked for later inspection. Giving a closer look to those files in white box penetration testing (Section 5.6.2) bore fruit: three different files used the same utility for handling references to files on the file system and this utility had a path traversal vulnerability.

This vulnerability was of the naiver kind; vulnerable code did not even check if the path was supposedly within allowed directory but instead gave an unrestricted access to any files on the system. Given that Resin normally runs as “local system” user (Section 6.6), access to all files was guaranteed.

These flawed parts of Efecte allowed deleting and downloading of arbitrary files, but only from the computer Efecte runs on. With all data residing in the database, hopefully on another machine, confidentiality was not impacted but integrity and availability were greatly compromised.

Explanation for this carelessness seemed to be that the request parameters were not from any user selectable part of user interface and thus were not meant for users to set. Instead, the parameters were used to pass information from one request to another which had led developers writing the code ignore the security issues.

6.5.2 Solution: a unified file access interface

Files are handled in multiple places in Efecte source code, creating multiple possibilities for failure.

The most secure solution would be to create an interface for handling files that would allow easy restriction of paths and file operations and to restrict management of files to only this trusted interface. Then, only one place of the code would need

to be trusted and testing effort could be spent only in one place leading to better testing and thus greater assurance of correct functioning would follow.

Unfortunately, this wide direct usage of files meant that there were plenty of places where change from old direct handling to indirect handling could cause bugs. For this reason, path traversal vulnerabilities were first fixed only in this one utility by canonizing the path before checking whether access to the path was allowed or not.

6.6 Efecte's extra privileges

Efecte, or more specifically Resin application server running Efecte, runs on a Windows machine as "local system" user, the Windows version of super-user. What Efecte actually needs is an ability to run a HTTP server on port 80 or 443 (for encrypted traffic), write to directory it is installed to and to initiate a connection to a database.

None of these operations need the vast access rights given to "local system" user in Windows, a normal limited user would suffice. Giving too large access rights does not create a vulnerability per se, but were Efecte to contain vulnerabilities affecting the computer it is run on, proper access rights would dampen the impact.

Same kind of violation of principle of least privilege exist in relation between Efecte and the database server: Efecte is given access rights of database administrator even though Efecte's normal function is just to write and read a small set of tables. Database administrator is allowed to perform any action to any table in the database: read, write and delete any table or its content.

Both of these weaknesses are consequences of making installation easy. Installing the Resin HTTP server Windows service as something other than "local system" would require creation of a limited user for Efecte use and installing the service in that user's name. Such an installation procedure would require modification of current Efecte installer and writing custom code in the installer as currently used installer builder does not support installation of services with a specific user, let alone creation of such users.

When Efecte is installed, Efecte needs to create the tables it will use in the database. This creation of tables is automated within the installer but requires database administrator access rights and instead of giving separate user name and password for installation and for normal usage, only one user name and password is used. Given that running Efecte then uses these administrative access rights and that companies rarely have dedicated databases for Efecte, effect of a vulnerability that allows access to database, such as SQL injection, is needlessly made more severe.

On top of these installer-based problems, the Inspector agent on workstations is run with "local system" privileges. The Inspector agent is a self-written C++ software that communicates through the network. There is a small chance of a buffer overflow vulnerability and thus running as a "local system" should be avoided if possible. For gathering most of the data and for listening to a port, the Inspector agent does

not require any privileges that a normal user does not have. The Inspector agent installation has the same problem as installing Resin HTTP server: it is easier to install a service as a "local system" instead of creating a limited user for running the agent.

6.6.1 Mitigation: reducing privileges

All these cases of too large privileges stem from the desire to make the installation of Efecte and accompanying tools easier. None of the described programs need to run with as large privileges as they are currently granted but circumstances force the situation. Although there are some claims that principle of least privilege is fundamentally wrong [Ber07], it is widely considered a good principle for reducing damages if not actual vulnerability.

Possible damages or risk exposure can be controlled in two ways: controlling the risk itself or by controlling the impact of the risk. Changing the Efecte and Inspector installers so that least required privileges are used would control the impact. Yet the impact of attacks such as SQL injection and path traversal is quite bad even without this additional exposure from the extra privileges so minimization of risk was chosen as the mitigation method.

Were creation of less privileged services to become sufficiently easy, this situation would get fixed, but for the time being, risk leverage of creating less privileged services is too low.

6.7 Loose access control in actions

As described in Section 5.2, requests to Efecte's JSP-based part are mapped to actions that are either Java classes or JSP files. Access control of actions is divided to two parts: configuration file governs whether or not an action requires authentication to Efecte and actions themselves combined with normal Efecte access controls govern what sort of permissions are required for an authenticated user.

There is actually only one action that does not require authentication and the controller wisely defaults to requiring authentication when unauthenticated access is not specifically allowed in configuration file. This defaulting to a choice that maintains integrity and confidentiality possibly at the expense of availability is the principle of fail-safe defaults, one of the eight principles discussed in Section 3.3.3.

Unfortunately, access to actions does not follow this principle. Instead, actions default to allowing access to all users that the controller has allowed to access the action meaning that if nothing else was specified in the action, only requirement was that user was authenticated. Denying access to actions is based on the explicit check of administrative permissions that developers need to add to each of the actions.

Other problem related to actions was that result pages assumed that control flew to them from actions but result pages could be reached on their own. As assumptions

form the basis of system's security, broken assumptions mean that system cannot be considered secure.

6.7.1 Evaluation

First problem originated from the lack of fail-safe defaults. Problem of not denying access to legitimate users (easy to notice) was turned into a problem of allowing access to illegitimate users (harder to notice). Ultimately, some actions required too few administrative permissions. Such actions allowed an attacker to read or modify information only an administrator was supposed to see, such as connected users or hit rate of a database cache.

These lapses of access control enabled attacks ranging from quite harmless disclosure of information to gaining of administrative privileges and thus complete loss of confidentiality, integrity and availability. There is no reason to implement access control in the described way of"opt-in" instead of following principle of fail-safe defaults.

The problem of broken assumptions meant that all of the result pages could cause security problems. As always with broken assumptions, this led to variety of problems covering confidentiality, integrity and availability.

Both problems were intensified by the fact that some actions and result pages act as trusted procedures: after checking for supposedly correct preconditions, they change user's level to "root", bypassing other security checks.

6.7.2 Solution: proper access control

A proper fix to the first problem is to implement access control of actions using by using fail-safe defaults; each action would require user's level to be root by default and lesser requirements could be added explicitly. Then an action editing users would require "edit users" administrative permission and if check for such permission was omitted, only a root could use the action.

Unfortunately, such a change of design affects naturally all actions and then some additional parts of Efecte. For this reason, such a change has not yet been implemented and instead old vulnerable actions have been modified to check for permissions correctly. As all actions were reviewed by hand during vulnerability analysis, we are assured that there are no flawed access checks in actions.

For the second problem, the problem of broken assumptions, there were two ways to fix the situation: add access checks to place where the actual access is done, not just to the start of an action (complete mediation) or make the assumption again valid. Making assumptions valid would require moving the result page JSP files to "WEB-INF", a directory that Java application servers deny direct access to and updating references to such JSP files. This is a solution that is easy to automate whereas complete mediation solution would require more work and duplicate some

of the checks both in the action and in the result page.

The problem was solved by moving the JSPs into “WEB-INF”. The solution has the downside that were the assumption become invalid again, as has been case earlier with Resin, the same vulnerabilities would follow.

7 Conclusions

Ten years is a long time on software industry, especially in the field of web applications. During that ten years, web technologies have advanced immensely and many applications that simply were impossible in 1999 are now used every day by millions of users. Unfortunately, new technologies have bought new attacks with them. Many of the attacks that were effective against Efecte were not even invented when development of Efecte started. During those then years of Efecte development, the development process at Efecte has changed from unstructured development to structured Scrum-process with proper code reviews, testing and education. Effects of this change were visible also in this work as many of the vulnerabilities found were old and could not get to the code base nowadays. Until this thesis' work, however, security issues were not substantially brought up.

This thesis represented efforts of a security push at Efecte similar to what Microsoft went through in 2002 [HL03], except in smaller scale. Work on the thesis started with self-education on the fundamentals of computer and information security. Fundamentals were followed by small parts from the field of computer security that seemed to make sense in the direction the thesis was going to although not all the knowledge found direct application in the empirical part.

What found direct use in the empirical part was all the research into web technologies, as knowing the problem domain is the first requirement for any successful analysis. Thus, different web technologies used to build web applications and common vulnerabilities in such applications were discussed and provided a good starting point to later analyses.

In the empirical part of this thesis, application of the security assessment methods suffered some reversals, partly for lack of proper tools, partly for lack of experience. Lack of tools made editing attack tree models cumbersome and proper attack surface analysis well nigh impossible. Still the result were, if not the best, at least good in the sense that new vulnerable areas in Efecte were uncovered. Most of the earlier work of Efecte security had been, rightly so, focused on common web application vulnerabilities and this thesis' work demonstrated that an application is not made secure just by fixing the most common vulnerabilities.

In a way, the security push is still ongoing. The knowledge gained in this thesis' theoretical and experimental part has yet to diffuse to other developers and not all devised enhancements for security are implemented nor will they all ever be because of the trade offs that need to made.

7.1 Future work

In this thesis, two possibly very useful methods, attack trees and attack surface analysis could not be applied in full because lack of tools and knowledge. Should there be another security push, revisiting those analysis methods and tools should be a starting point.

Two large stones that were left completely unturned were use of static analysis tools and of other programming languages.

Static analysis tools, such as CodeSecure based on WebSSARI [HYH+04], analyze the code of the application without actually running it, hoping to find signs of vulnerabilities such as insecure flow of data or unchecked memory accesses. If such tool could be integrated seamlessly into development environment, benefits would be great. Warning of security errors as modern development environments warn of syntax errors is one thing but more importantly, such tool could most certainly analyze security issues in third party libraries used.

Different programming languages could offer at least two benefits: better specification of levels of trust for the data and a possibility to separate security concerns from other concerns, making parts of a program easier to understand and use.

Ultimately, none of these methods and tools will be a silver bullet for the slaying beast of security problems. Still, sometimes even grazing it will be worthwhile.

References

[AAMS03] Philip S. Antón, Robert H. Anderson, Richard Mesic, and Michael Scheiern, *Finding and Fixing Vulnerabilities in Information Systems: The Vulnerability Assesment & Mitigation Methodology*, Tech. report, RAND Corporation, 2003.

[And01] Ross J. Anderson, *Security Engineering: a guide to building dependable distributed systems*, Wiley Computer Publishing, 2001.

[AY08] Charu C. Aggarwal and Philip S. Yu (eds.), *Privacy-Preserving Data Mining: models and algorithms*, Advances in Database Systems, Springer US, 2008.

[BC08] Eli Biham and Yaniv Carmeli, *Efficient Reconstruction of RC4 Keys from Internal States*, Springer-Verlag, Berlin, Heidelberg, 2008.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts, *Stateful Traits*, LNCS, vol. 4406, Springer, August 2007.

[Ber07] Daniel J. Bernstein, *Some thoughts on security after ten years of qmail 1.0*, CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture (New York, NY, USA), ACM, 2007, pp. 1–10.

[Bib75] K. J. Biba, *Integrity Considerations for Secure Computer Systems Technical Report MTR-3153*, Tech. report, MITRE Corporation, June 1975.

[Bis02] Matt Bishop, *Computer Security : art and science*, Pearson Education, 2002.

[Bis07] M. Bishop, *About Penetration Testing*, Security & Privacy, IEEE **5** (2007), no. 6, 84–87.

[BL73] D. E. Bell and L. J. La Padula, *Secure Computer Systems: Mathematical Foundations*, Tech. Report Technical Rerpot MTR-2547, MITRE Corporation, 1973.

[Bro87] Jr. Brooks, F.P., *No Silver Bullet: Essence and Accidents of Software Engineering*, Computer **20** (1987), no. 4, 10–19.

[Chr09] Steve Christey, *2009 CWE/SANS Top 25 Most Dangerous Programming Errors version 1.1*, Website, March 2009, `http://cwe.mitre.org/top25/pdf/2009_cwe_sans_top_25.pdf`.

[CLR+00] S. Cavallar, W. M. Lioen, H. J. J. Te Riele, B. Dodson, A. K. Lenstra, P. L. Montgomery, B. Murphy Et Al, Mathematisch Centrum (smc, The Dutch Foundation, Stefania Cavallar, Walter Lioen, Herman Te Riele, Bruce Dodson, Arjen K. Lenstra, Peter L. Montgomery, Brian

Murphy, Karen Aardal, Je Gilchrist, and Gerard Guillerm, *Factorization of a 512-bit RSA modulus*, pp. 1–18, Springer-Verlag, 2000.

[CW87] David D. Clark and David R. Wilson, *A Comparison of Commercial and Military Computer Security Policies*, Security and Privacy, IEEE Symposium on **0** (1987), 184 – 194.

[DA99] T. Dierks and C. Allen, *The TLS Protocol Version 1.0*, Website, January 1999, `http://tools.ietf.org/html/rfc2246`.

[DJL79] David Dobkin, Anita K. Jones, and Richard J. Lipton, *Secure databases: protection against user influence*, ACM Trans. Database Syst. **4** (1979), no. 1, 97–106.

[Fai05] Richard E. Fairley, *Software Risk Management*, IEEE Software **22** (2005), no. 3, 101.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, Website, June 1999, `http://tools.ietf.org/html/rfc2616`.

[Fie00] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. thesis, Dept. of Information and Computer Science, University of California, 2000.

[Gar05] Jesse James Garrett, *Ajax: A New Approach to Web Applications*, Website, 2005, `http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[Gas88] Morrie Gasser, *Building a secure computer system*, Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[GGKL89] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, *The Digital Distributed System Security Architecture*, Proc. 12th National Computer Security Conf, 1989, pp. 305 – 319.

[GM05] Zvi Gutterman and Dahlia Malkhi, *Hold Your Sessions: An Attack on Java Session-Id Generation*, CT-RSA, 2005, pp. 44–57.

[Gol01] Dieter Gollmann, *Security Protocols*, ch. Mergers and Principals, pp. 5 – 13, Springer Berlin / Heidelberg, 2001.

[Gol05] ———, *Computer security*, 2nd ed., John Wiley & Sons, Inc., New York, NY, USA, 2005.

[GPP+06] Xiaocheng Ge, Richard F. Paige, Fiona A.C. Polack, Howard Chivers, and Phillip J. Brooke, *Agile development of secure web applications*, ICWE '06: Proceedings of the 6th international conference on Web engineering (New York, NY, USA), ACM, 2006, pp. 305–312.

[Har88] Norm Hardy, *The Confused Deputy: (or why capabilities might have been invented)*, SIGOPS Oper. Syst. Rev. **22** (1988), no. 4, 36–38.

[HL02] Michael Howard and David E. Leblanc, *Writing Secure Code*, Microsoft Press, Redmond, WA, USA, 2002.

[HL03] M. Howard and S. Lipner, *Inside the Windows security push*, Security & Privacy, IEEE **1** (2003), no. 1, 57–61.

[HL06] Michael Howard and Steve Lipner, *The Security Development Lifecycle*, Microsoft Press, Redmond, WA, USA, 2006.

[Hop07] Alex Hopmann, *The story of XMLHTTP*, Website, 2007, `http://www.alexhopmann.com/xmlhttp.htm`.

[HPW03] Michael Howard, Jon Pincus, and Jeannette M. Wing, *Measuring Relative Attack Surfaces*, Proceedings of Workshop on Advanced Developments in Software and Systems Security, 2003.

[Hug89] J. Hughes, *Why functional programming matters*, Comput. J. **32** (1989), no. 2, 98–107.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo, *Securing web application code by static analysis and runtime protection*, WWW '04: Proceedings of the 13th international conference on World Wide Web (New York, NY, USA), ACM, 2004, pp. 40–52.

[ISO] ISO/IEC, *ISO/IEC 15408-1 information technology - security techniques - evaluation criteria for IT security.*

[Kil90] H. Kilov, *From semantic to object-oriented data modeling*, Systems Integration, 1990. Systems Integration '90., Proceedings of the First International Conference on (1990), 385–393.

[Kle90] D. V. Klein, *Foiling the Cracker: A Survey of, and Improvements to, Password Security*, "nd USENIX Unix Security Workshop, 1990, pp. 5 – 14.

[Kle05] Amit Klein, *DOM based cross site scripting or XSS of the third kind*, Website, April 2005, `http://www.webappsec.org/projects/articles/071105.shtml`.

[KM97] D. Kristol and L. Montulli, *HTTP State Management Mechanism*, Website, February 1997, `http://tools.ietf.org/html/rfc2109`.

[KM00] _____, *HTTP State Management Mechanism*, Website, October 2000, `http://tools.ietf.org/html/rfc2965`.

[Kon06] Vidar Kongsli, *Towards agile security in web applications*, OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM, 2006, pp. 805–808.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, *Authentication in distributed systems: theory and practice*, ACM Trans. Comput. Syst. **10** (1992), no. 4, 265–310.

[Lam71] Butler W. Lampson, *Protection*, Proc. Fifth Princeton Symposium on Information Sciences and Systems, 1971, pp. 437–443.

[Lip82] Steven B. Lipner, *Non-Discretionary Controls for Commercial Applications*, IEEE Symposium on Security and Privacy, 1982, pp. 2–10.

[Mar07] Robert A. Martin, *Being Explicit About Security Weaknesses*, CrossTalk The Journal of Defense Software Engineering (2007), 4–8.

[McG06] Gary McGraw, *Software Security: building security in*, Addison-Wesley, 2006.

[MDW02] M. Mealling, R. Denenberg, and W3C URI Interest Group, *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*, Website, April 2002, `http://tools.ietf.org/html/rfc3305`.

[MF99] J. McDermott and C. Fox, *Using abuse case models for security requirements analysis*, Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual (1999), 55–64.

[MKW08] Pratyusa K. Manadhata, Yuecel Karabulut, and Jeannette M. Wing, *Measuring the Attack Surfaces of SAP Business Applications*, Tech. Report CMU-CS-08-134, Carnegia-Mellon University, 2008.

[MO05] Sjouke Mauw and Martijn Oostdijk, *Foundations of Attack Trees*, International Conference on Information Security and Cryptology – ICISC 2005. LNCS 3935, Springer, 2005, pp. 186–198.

[MSR+07] Peter Mell, Karen Scarfone, Sasha Romanosky, Interest Group Members, Including Barrie Brook, Seth Hanford, Stav Raviv, Gavin Reid, and George Theall, *A Complete Guide to the Common Vulnerability Scoring System Version 2.0*, Website, June 2007, `http://www.first.org/cvss/cvss-guide.html`.

[MTMW07] Pratyusa K. Manadhata, Kymie M. C. Tan, Roy A. Maxion, and Jeannette M. Wing, *An Approach to Measuring A System's Attack Surface*, Tech. report, School of Computer Science Carnegie Mellon University, August 2007, `http://reports-archive.adm.cs.cmu.edu/anon/2007/CMU-CS-07-146.pdf`.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Inc., 2001.

[PF02] Stephen R. Palmer and John M. Felsing, *A Practical Guide to Feature-Driven Development (The Coad Series)*, Prentice Hall PTR, February 2002.

[PP06] Charles P. Pfleeger and Shari Lawrence Pfleeger, *Security in Computing*, 4th ed., Pearson Education, 2006.

[Res00] E. Rescorla, *HTTP Over TLS*, Website, May 2000, `http://tools.ietf.org/html/rfc2818`.

[Ric] Robert Richardson, *CSI Computer Crime & Security Survey 2008*, website.

[Rit07] Paul Ritchie, *The security risks of AJAX/web 2.0 applications*, Network Security **2007** (2007), no. 3, 4–8.

[Ros03] Ronald G. Ross, *Principles of the Business Rule Approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[RTJ00] Dirk Riehle, Michel Tilman, and Ralph Johnson, *Dynamic Object Model*, Tech. report, SKYVA International, UNISYS Belgium, Computer Science Department University of Illinois at Urbana-Champaign, 2000.

[SBK05] M. Siponen, R. Baskerville, and T. Kuivalainen, *Integrating Security into Agile Development Methods*, System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on (2005), 185a–185a.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, *Role-based access control models*, Computer **29** (1996), no. 2, 38 – 47.

[Sch96] Bruce Schneier, *Applied Cryptography*, 2nd ed., John Wiley & Sons, 1996.

[Sch99] ———, *Attack Trees*, Dr. Dobb's Journal (1999), 21 – 29.

[SO05] Guttorm Sindre and Andreas L. Opdahl, *Eliciting security requirements with misuse cases*, Requirements Engineering **10** (2005), 34 – 44.

[SS75] J.H. Saltzer and M.D. Schroeder, *The protection of information in computer systems*, Proceedings of the IEEE **63** (1975), no. 9, 1278–1308.

[SS04] Frank Swiderski and Window Snyder, *Threat Modeling*, Microsoft Press, 2004.

[SSA+09] Marc Stevens, Alex Sotirov, Jake Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger, *Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate*, Cryptology ePrint Archive, Report 2009/111, 2009, `http://eprint.iacr.org/`.

[Sti06] Doug Stinson, *Cryptography Theory and Practice*, 3rd ed., CRC Press, Inc., 2006.

[Tho03] H.H. Thompson, *Why security testing is hard*, Security & Privacy, IEEE **1** (2003), no. 4, 83–86.

[Uni99] United Nations, *International Review of Criminal Policy - United Nations Manual on the Prevention and Control of Computer-related Crime*, Website, 1999, `http://www.uncjin.org/Documents/EighthCongress.html`.

[vdSWW07] Andrew van der Stock, Jeff Williams, and Dave Wichers, *The ten most critical web application security vulnerabilities for JAVA enterprise applications*, website, 2007, `https://www.owasp.org/images/8/89/OWASP_Top_10_2007_for_JEE.pdf`.

[VHS03] Jr. Vaughn, R.B., R. Henning, and A. Siraj, *Information assurance measures and metrics - state of practice and proposed taxonomy*, System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on (2003), 10 pp.–.

[wg] ES4 working group, *Proposed ECMAScript 4th Edition - Language Overview*, Website, `http://www.ecmascript.org/es4/spec/overview.pdf`.

[Wha01] Sean Whalen, *An Introduction to ARP Spoofing*, `http://www.rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf`, April 2001, Website.

[YBAG04] J. Yan, A. Blackwell, R. Anderson, and A. Grant, *Password memorability and security: empirical results*, Security & Privacy, IEEE **2** (2004), no. 5, 25–31.

[Zal05] Michał Zalewski, *Silence on the Wire*, No Starch Press, 2005.

[Zal09] ———, *Browser Security Handbook*, Website, April 2009, `http://code.google.com/p/browsersec/wiki/Main` fetched 13.5.2009.